COORDINATED HIGHWAYS ACTION RESPONSE TEAM
STATE HIGHWAY ADMINISTRATION

R1B2 Servers Detailed Design

Contract DBM-9713-NMS
TSR # 9901961
Document # M362-DS-006R0

May 26, 2000
By
Computer Sciences Corporation and PB Farradyne Inc.

Revision	Description	Pages Affected	Date
0	Initial Release	All	May 26, 2000

Table of Contents

Table of Figures

1 Introduction

1.1 Purpose

This document describes the detailed design of the CHART II system software for Release 1, Build 2. This design is driven by the Release 1, Build 2 requirements as stated in document M361-002R1, "***CHART II System Requirements Specification Release 1 Build 2***" and further refines the high level design presented in document M362-DS-005, ***"R1B2 High Level Design."***

1.2 Objectives

The main objective of this design is to provide software developers with details regarding the implementation of the service applications used to satisfy the requirements of Release 1, Build 2 of the CHART II system.

This design also serves to provide documentation to those outside of the software development community to show how the requirements are being accounted for in the software design.

1.3 Scope

This design is limited to Release 1, Build 2 of the CHART II system and the requirements as stated in the aforementioned requirements document that have been allocated to Release 1, Build 1 or Build 2. Additionally, this design document includes only the design of CHART II services and does not include the design of the Graphical User Interface, Database Schema, or Field Communications.

1.4 Design Process

As in the high level design, object-oriented analysis and design techniques were used in creating this design. As such, much of the design is documented using diagrams that conform to the Unified Modeling Language (UML), a de facto standard for diagramming object-oriented designs.

In the high level design, system interfaces were identified and specified. These interfaces were partitioned into logical groupings of packages. This design serves to fill in the details necessary to implement each of the system interfaces identified in the high level design.

In this design, each package identified in the high level design is addressed separately with its own class diagram and sequence diagrams for major operations included in the package's interfaces. Additionally, packages needed for implementation but not present in the high level design are included in this design, with each of these also having its own class diagram and sequence diagrams. Packages are also included for third party software that is needed by the CHART II software, such as the ORB and Java classes. Only classes and methods shown on the sequence diagrams are included in diagrams for third party products.

The design process for each package involved starting with a class diagram including interfaces from the high level design, and filling in details to the class diagram to move toward implementation. Sequence diagrams were then used to show how the functionality is to be carried out. An iterative process was used to enhance the class diagram as sequence diagrams identified missing classes or methods.

1.5 Design Tools

The work products contained within this design are extracted from the COOL:JEX design tool. Within this tool, the design is contained in the CHART II project, R1B2 configuration, System Design phase. A system version is included for each software package.

1.6 Work Products

This design contains the following work products:

- A UML Class diagram for each package showing the low level software objects which will allow the system to implement the interfaces identified in the high level design.
- UML Sequence diagrams for non-trivial operations of each interface identified in the high level design. Additionally, sequence diagrams are included for non-trivial methods in classes created to implement the interfaces. Operations that are considered trivial are operations that do nothing more than return a value or a list of values and where interaction between several classes is not involved.

2 Key Design Concepts

This section discusses various elements of the design that warrant more discussion than the UML diagrams afford. The High Level Design Document referenced above provides background information on CORBA and R1B2 Packaging and Deployment that may be necessary to fully benefit from the discussions below.

2.1 Access Control

As discussed in the R1B2 High Level Design, the CHART II system uses a flexible access control system based around the following basic elements:

- users
- system functions
- shared resources
- functional rights and roles.

Each user of the system is assigned one or more roles. Each role has one or more functional rights. Each system function must ensure the user initiating the operation has the proper functional right before allowing the function to be executed. Shared Resources, which have an owning organization, utilize an organization filter in conjunction with certain functional rights to allow rights to be granted based on the organization that owns the device. For example, a role may be granted a functional right to set a message on SHA DMSs but not MDTA DMSs.

This design allows access to groups of system functions to be assigned to users to easily provide each user with the desired level of access to the system. The table in Appendix A shows each of the system functions in R1B2 for which access control is supplied. Also shown in the table is the functional right that is required for a user to execute the system function and whether the functional right can be used in conjunction with an organization filter.

Implementations of system functions rely on two key elements to carry out access control, a user access token and a token manipulator. When a user logs into the system, the UserManager object returns an access token to the GUI that contains a binary encoding of the functional rights that are held by the user, as defined by their currently assigned roles.

When the user attempts to execute a system function that is access controlled, the GUI passes the user’s token as a parameter to the system function. Each system function is coded to know exactly which functional right is required to execute the function (see the table in Appendix A below). To determine if the user should be allowed to execute a system function, the system function passes the user’s access token and the function’s required functional right to an object called a token manipulator, which tells the function if it should allow execution or not.

The token manipulator encapsulates the knowledge of the binary format of an access token and keeps the burden of access control minimal for system functions.

2.2 Operations Logging

The CHART II system tracks all usage of access controlled system functions through the operations log. When a user successfully executes such a function, a record is stored in the operations log table in the CHART II database that contains the user's name, operations center, date and time, a description of the operation the user performed, and a category for the operation. To ease the burden on system functions in performing this task, an OperationsLog utility class exists. This utility class provides an API that allows an entry to be added to the operations log without the system function having to interface with the database directly.

Although every access controlled system function utilizes the OperationsLog class to perform operations logging, many diagrams in section 3 below do not show this class interaction due to the limited amount of space available on each diagram.

2.3 Service Application Framework

In a CORBA based system, service applications are used to serve CORBA objects through the ORB, making them available for use by other applications through a network. Once an object has been created and connected to the ORB, the object can act as an independent piece of software, given access to some basic services. The service applications that are built to serve CORBA objects usually share the same basic structure and functionality. The design team took advantage of this fact to provide a reusable framework for service applications.

The design of the application framework for CHART II CORBA Services is based upon two interfaces, the ServiceApplication and the ServiceApplicationModule. A class that implements the ServiceApplication interface is able to provide the basic services needed by CHART II CORBA objects. A ServiceApplicationModule is responsible for the initialization and shutdown of specific CORBA objects, using the services provided by the ServiceApplication.

Several classes that implement the ServiceApplicationModule interface are included in this design, with each module responsible for serving one or more specific CHART II CORBA classes. Each of these modules has its own initialization and shutdown methods tailored to the needs of the objects that it serves. Typical module initialization involves object creation from a state persisted in the database, connecting objects to the ORB, creation of an event channel, and publication of objects in the Trading Service. Typical module shutdown involves disconnecting objects from the ORB and destroying the objects.

The DefaultServiceApplication class provides a default implementation of the ServiceApplication interface. The DefaultServiceApplication is capable of hosting one or more ServiceApplicationModules. A configuration file used by the DefaultServiceApplication specifies the modules served by a specific instance of the DefaultServiceApplication. This design allows for flexibility in the partitioning of objects among software processes. Modules can be brought together into a single process to achieve performance gains or moved to separate processes to provide greater fault isolation.

The design of the Service Application Framework is evidenced throughout this design. Packages exist for each module and a package named CHART2Service provides an application entry point for the DefaultServiceApplication.

2.4 Service Application Maintenance

The CHARTService application implements the Service interface (defined in IDL) to allow for clean service shutdown. In addition to allowing shutdown, the Service interface includes features that will be useful for a future system monitor process. These features include the ability for a service to tell its name when asked, tell the network connection site where it is running, and respond to a ping operation. Since the Service is a CORBA object attached to an ORB, these operations on a service can be accessed from anywhere on the CHART II network.

2.5 Event Channel Fault Tolerance

The standard CORBA event service contains a single event channel that is accessed through transient objects served by the event service called consumers and suppliers. Since the objects are transient, if the event service should crash, applications using the event service need to reinitialize their connection to the event service once it becomes available. The CHART II R1B2 design contains utility classes that allow applications to be tolerant of restarts of the event service. The PushEventSupplier, PushEventConsumer, and EventConsumerGroup classes, and the EventConsumer interface provide functionality for maintaining the connection to an event channel. The PushEventSupplier works as a wrapper to a CORBA PushSupplier that detects when an attempt to push fails and automatically attempts to reconnect on subsequent pushes.

The EventConsumer and EventConsumerGroup work together to allow multiple associations of event channels and consumers to be maintained, with a polling thread that periodically checks the connection of the consumer to the event channel and performs an automatic reconnect if necessary. The PushEventConsumer is an implementation of the EventConsumer that uses the push event model.

In addition to the need to provide fault tolerance for the CORBA Event Service, the standard event service's limitation to a single event channel causes events of all types to be passed on the same event channel. While this provides no hardship to suppliers of events, it requires consumers to filter the events to determine if they need to take action on an event or throw it away. This leads to inefficiency in both the processing required to filter the events as well as the network bandwidth used to pass unwanted events to consumers. This also makes it harder to provide a modular GUI design that allows seamless addition of new functionality.

To make up for this shortcoming, this design makes use of the ORB vendor's extension to the event service that includes an EventChannelFactory interface that provides the capability for creating multiple event channels within a single EventService. The CHART II R1B2 design utilizes this added functionality to allow each module to be responsible for creating an event channel in their local event service and publishing the event channel object in the trader. This allows event channels throughout the system to be collected to provide a "big picture" of the real time status of the system and also provides fault isolation if an event service should fail.

2.6 Object Publication

As discussed in the High Level Design, the CORBA Trading Service is used by CHART II to allow CORBA objects to be discovered and used by other applications, including the CHART II GUI. All objects published in the Trading Service from CHART II applications are published with a service type equal to the interface name which the object implements. Full interface name hierarchies are used through the use of the supertypes registration feature (such as SharedResource / DMS) to allow generic as well as specific queries. All CHART II objects published in the trader have a standard mandatory property named "ID" of type octet sequence. This ID is a globally unique identifier that remains with the object for the life of the object, even through multiple restarts of the service serving the object. Use of this ID allows objects to be located regardless of where they are being served in the system.

The following CHART II R1B2 objects are published in the Trading Service:

- CHART2DMS
- CHART2DMSFactory
- CHART2HAR
- CHART2HARFactory
- CommLog
- Dictionary
- EventChannel
- LibraryFactory
- MessageLibrary
- Organization
- Plan
- PlanFactory
- SHAZAM
- SHAZAMFactory
- StoredMessage
- TrafficEvent
- TrafficEventFactory
- UserManager

2.7 Pass By Value

Some system interfaces in this design rely on the pass by value feature of CORBA 2.3. Pass by value allows a copy of a software object created by a client to be passed as an argument to a CORBA servant (or vice versa). While this concept is much like passing a group of values between CORBA servant and client as a structure, it features the ability to use subclassing to allow the objects to behave polymorphically.

An example of the use of pass by value in this design is evident in the DMS control interfaces. A value type named DMSStatus is defined which contains status values that are common to all DMS devices. CHART2DMSStatus adds status values specific to CHART II, such as the controlling operations center of the device. Model specific derivations add status values only present in specific DMS models, such as the error status bits of an FP9500.

This use of subclassing allows the DMS interface to specify a method named getStatus() that returns a DMSStatus object. The specific implementation of the DMS object will pass back the appropriate “flavor” of DMSStatus based on the DMS model. As new DMS models are added to the system, the interface does not change, which means previously developed code can remain stable.

Subclassing of Status objects also allows a GUI that encounters a sign model for which it does not have a model specific status dialog to show status information that is defined in the base class, DMSStatus. While this is of no benefit for sign models coded for directly under the CHART II project, it would allow DMS objects published by other organizations to be viewed easily by the CHART II GUI, without the CHART II GUI having to add any code specific to the sign model.

2.8 Database Access

A relational database is used to store system configuration data, persist object states (to allow restarts to assume their previous state), and to log user operations in the operations log. Java Database Connectivity (JDBC) is used within the application software to access the database. Access to the database is managed by the CHART II DBConnectionManager class. This class manages connections to the database. Each software package that requires access to the database includes a class that contains methods for all database accesses needed by the package. These classes are named with the package name and a suffix of DB. These database classes all use the DBConnectionManager object to obtain a JDBC connection to the database each time a series of queries or statements are to be executed. By managing a pool of actual database connections, the DBConnectionManager class makes sure that only one thread at a time has access to a given database connection, thus allowing transactional processing to be done safely.

2.9 Field Communications

Field communications are necessary in R1B2 to control DMS, HAR, and SHAZAM devices. The design for field communications is provided by the FMS subsystem and is not included in this design. This design includes placeholder objects used to show the interface points with the FMS communications system. These objects are only placeholders at this time because the FMS detailed design is not complete at the time of this writing. The FMS detailed design will contain a full discussion on the interface provided to the CHART II system.

2.10 Error Processing

Because CHART II is a distributed object system, it is expected that any call to a remote object could cause a CORBA exception to be thrown. All software calls to remote objects handle CORBA exceptions and the processing is not shown on sequence diagrams within this design except where it serves to illustrate a design point.

Furthermore, as with any system, most method calls, system calls, etc. can fail unexpectedly. All such errors are handled by the software and are not shown explicitly in the package design portion of this document. The default action when such an error is encountered is to reach a consistent state within the object where the error occurred and then to throw a CHART2Exception (even for non-CORBA calls). The CHART2Exception contains debugging information as well as text suitable for display to a user or administrator. These exceptions are shown on sequence diagrams to call out error conditions that are not obvious.

The Log utility class is used by modules to log error conditions to a flat file that is created by the service application hosting the module. The log file entries contain the name of the class that logged the entry, the date and time of the entry, and descriptive text of the error that occurred. The Log utility also provides the capability for a stack trace to be printed to the file to accompany the error. This feature is reserved for use when an error condition is caught and the exact cause of the error condition is not known. Log files created by the Log utility class are self-cleaning and are automatically removed from the system when they reach a certain age, as specified in a configuration file.

2.11 Recorded Voice Handling

This design accounts for the ability for operators to record voice at their workstation for broadcast on a HAR device. Because voice data can be very large, the passing of this voice data is minimized through the use of wrapper objects and streamers.

Recorded voice is supported in the CHART II system for

- immediate broadcast on a HAR
- storage in a slot on a HAR for future broadcast, and
- storage in a message library.

When voice is recorded the voice data is packaged in a HARMessageAudioDataClip object, which in turn is included in a HARMessage object. Upon receiving a HARMessageAudioDataClip, the CHART2HAR or MessageLibraryDB objects use a utility class named HARAudioClipManager to persist the audio data and obtain a HARMessageAudioClip in place of the HARMessageAudioDataClip. The HARMessageAudioClip contains a unique ID assigned to the voice data and a reference to an object known as a streamer that can provide access to the actual voice data given the ID. In CHART II R1B2, the HARAudioClipManager utility is a streamer and places a reference to itself in every HARMessageAudioClip it creates.

Because HARMessageAudioClip objects are small, they can be passed throughout the system as the part of the device status for a HAR without having a significant impact on network bandwidth usage. The only times the recorded voice data will be passed across the network after its initial storage will be when the user wishes to listen to the voice data or the voice needs to be recorded onto the HAR device. When this occurs, the HARMessageAudioClip is told to stream the data and the HARMessageAudioClip delegates the request to the streamer reference it contains, which is always the HARAudioClipManager where the data was originally stored.

Recorded voice data is automatically cleaned up from the servers when it is no longer needed. CHART2HAR objects request that their HARAudioClipManager delete the voice data when an immediate message containing a HARMessageAudioClip is blanked or replaced by a different message, or when a slot containing a HARMessageAudioClip is deleted or replaced. When a StoredMessage is deleted from the system, the MessageLibraryDB object requests that any HARMessageAudioDataClips contained in a HARMessage be deleted from the system. An owner ID is used by the HARAudioClipManager to distinguish clips stored by the message library vs. clips stored by a HAR. This is necessary because the CHART2HAR objects indiscriminately ask their HARAudioClipManager to delete voice data associated with any HARMessageAudioClip they are through playing. The owner ID is used to keep the CHART2HAR from deleting a clip that is part of a stored message.

2.12 Packaging

This software design is broken into many packages of related classes. The table below shows each of the packages along with a description of each.

CHART2Service	This package contains an implementation of the ServiceApplication interface specified in the utilities package. This implementation is used as the base application for serving one or more service application modules. Configuration files are used to configure the service application to specify the service application modules that will run within an instance of the application.
CommLogModule	This package contains a service application module that serves the CommLog interface as specified in the system interfaces.
CORBAUtilities	This package contains classes included in the third party ORB product used for implementation. Only classes that are directly referenced from diagrams for CHART II software are included in this package's diagrams.
DeviceUtility	This package contains utility classes that are shared device packages, such as DMS and HAR. This includes an implementation of the arbitration queue.
DictionaryModule	This package contains a service application module that serves the Dictionary interface as specified in the system interfaces.
DMSControlModule	This package contains a service application module that serves the CHART2DMSFactory and CHART2DMS objects as specified in the system interfaces.

DMSUtility	This package contains utility classes that are shared among the server and GUI DMS modules. Examples of DMSUtility classes are the MultiConverter and implementation of value types defined in the DMSControl system interfaces.
HARControlModule	This package contains a service application module that serves the CHART2HAR and CHART2HARFactory interfaces.
HARUtility	This package contains HAR related utility classes shared by the server and GUI.
JavaClasses	This package contains classes included in the Java programming language. Only classes that are directly referenced from diagrams for CHART II software are included in this package's diagrams.
MessageLibraryModule	This package contains a service application module that serves the LibraryFactory, MessageLibrary, and StoredMessage interfaces specified in the system interfaces.
PlanModule	This package contains a service application module that serves the PlanFactory, Plan, and Plan Item interfaces specified in the system interfaces.
ResourcesModule	This package contains a service application module that serves the OperationsCenter and Organization interfaces specified in the system interfaces.
SHAZAMControlModule	This package contains a service application module that serves SHAZAM and SHAZAMFactory interfaces as specified in the system interfaces.
SHAZAMUtility	This package contains SHAZAM related utility classes shared by the server and GUI.
SystemInterfaces	This package contains the CORBA interfaces and related definitions for the CHART II system. These interfaces and classes define the IDL for the CHART II system.

TrafficEventModule	This package contains a service application module that serves instances of the TrafficEvent interface as specified in the system interfaces.
TTSControlModule	This package contains a service application module that serves the TTSControl interface as specified in SystemInterfaces. This interface provides conversion from text to speech.
UserManagementModule	This package contains a service application module that serves the UserManager interface specified in the system interfaces.
Utility	This package contains utility classes shared by other packages, including classes used to access the database and the OperationsLog class.

The remainder of this document contains detailed designs of each of the above packages.

3 Package Designs

The following sections provide detailed designs of each of the software packages included in CHART II R1B2. Each section contains a class diagram and sequence diagrams for non-trivial operations for a software package.

3.1 CHART2Service

3.1.1 Classes

3.1.1.1 CHART2ServiceClasses (Class Diagram)

The diagram shows classes of an application that helps in installation and termination of the modules related to CHART II system.

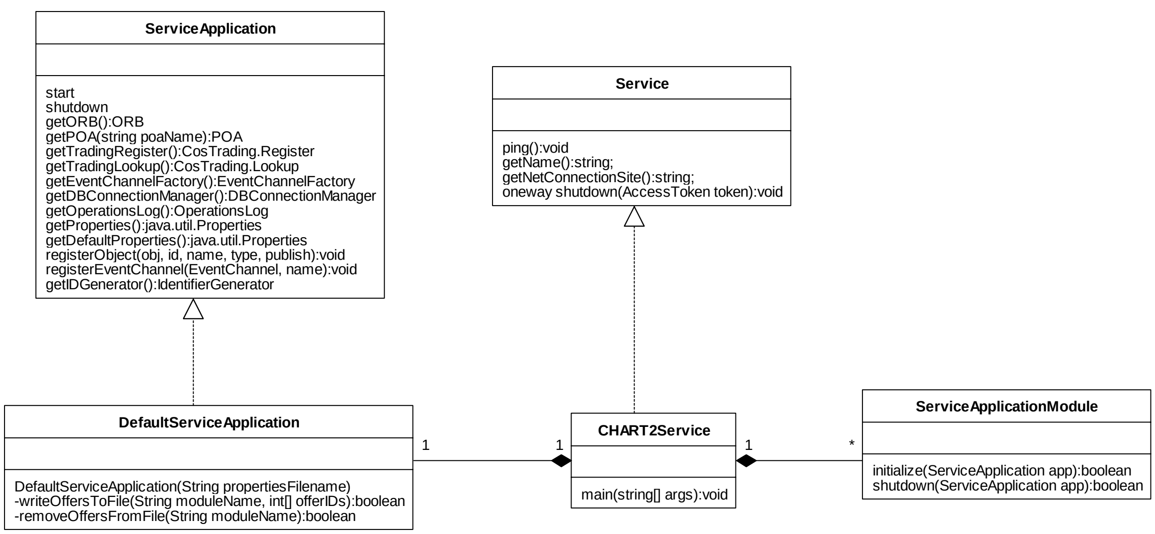

Figure 1. CHART2ServiceClasses (Class Diagram)

3.1.1.1.1 CHART2Service (Class)

The CHART2Service is an application that helps in installation and termination of the modules in CHART II system.

3.1.1.1.2 DefaultServiceApplication (Class)

This class is the default implementation of the ServiceApplication interface. This class is passed a properties file during construction. This properties file contains configuration data used by this class to set the ORB concurrency model, determine which ORB services need to available, provide database connectivity, etc. The properties file also contains the class names of service modules that should be served by the service application. During startup, the DefaultServiceApplication instantiates the service application module classes listed in the properties file and initializes each.

The DefaultServiceApplication maintains a file of offers that have been exported to the Trading Service. Each module must provide an implementation of the getOfferIDs method and be able to return the offer IDs for each object they have exported to the trader during their initialization. The DefaultServiceApplication stores all offer IDs in a file during its startup. Each module is expected to remove its offers from the trader during a shutdown. If the DefaultServiceApplication is not shutdown properly, it uses its offer ID file to clean-up old offers prior to initializing modules during its next start. This keeps multiple offers for the same object from being placed in the trader.

3.1.1.1.3 Service (Class)

This interface is implemented by all services in the system that allow themselves to be shutdown externally. All implementing classes provide a means to be cleanly shutdown and can be pinged to detect if they are alive.

3.1.1.1.4 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.1.1.1.5 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.1.2 Sequence Diagrams

3.1.2.1 CHART2Service:Shutdown (Sequence Diagram)

This sequence diagram shows shutdown of CHART2Service. This service calls shutdown on DefaultServiceApplication object that shuts down the modules that are served by the CHART II system. Refer to DefaultServiceApplication's Shutdown sequence diagram in Utility package for details. The CHART2Service deactivates itself using the POA and the CHART2Service calls the deactivate method on the POAManager to exit the event loop and shudown.

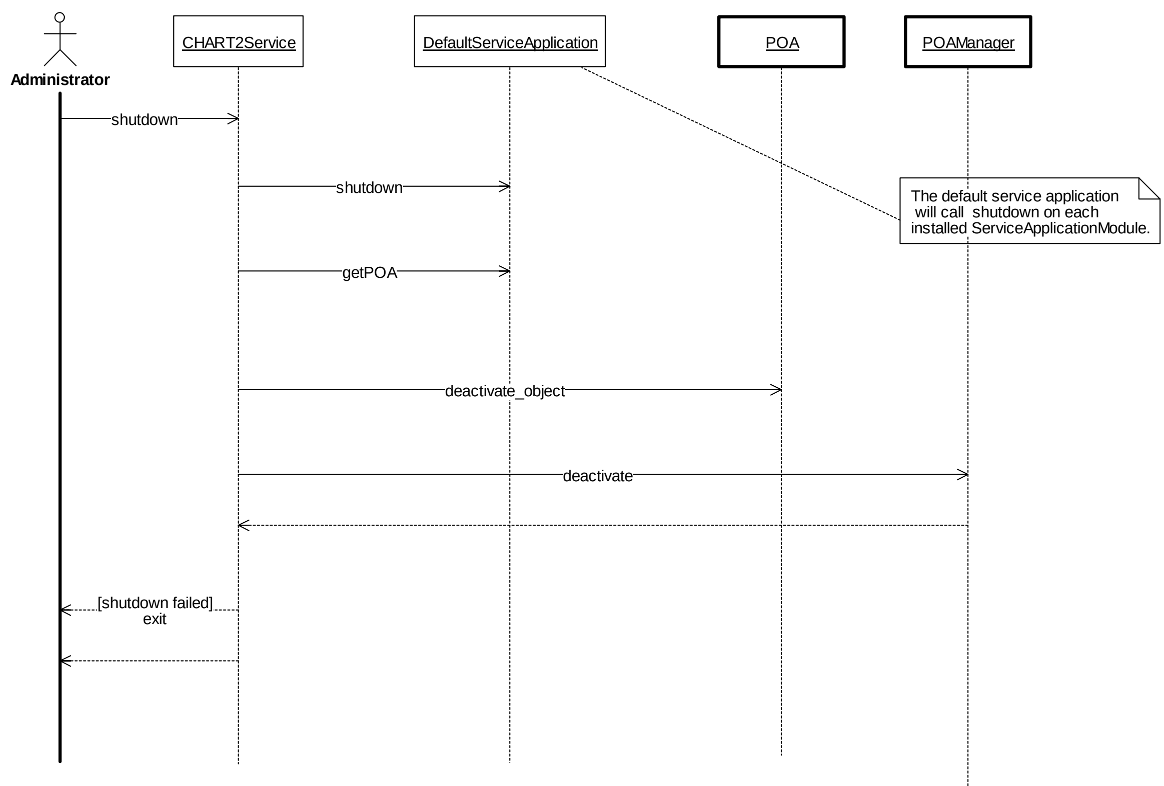

Figure 2. CHART2Service:Shutdown (Sequence Diagram)

3.1.2.2 CHART2Service:Startup (Sequence Diagram)

This sequence diagram shows startup of CHART2Service. This service creates and starts a DefaultServiceApplication object and the modules that are served by the CHART II system. Refer to DefaultServiceApplication's Start sequence diagram in Utility package for details. The CHART2Service is activated using the POA and the CHART2Service activates the POAManager to enter the event loop and start serving the CORBA requests.

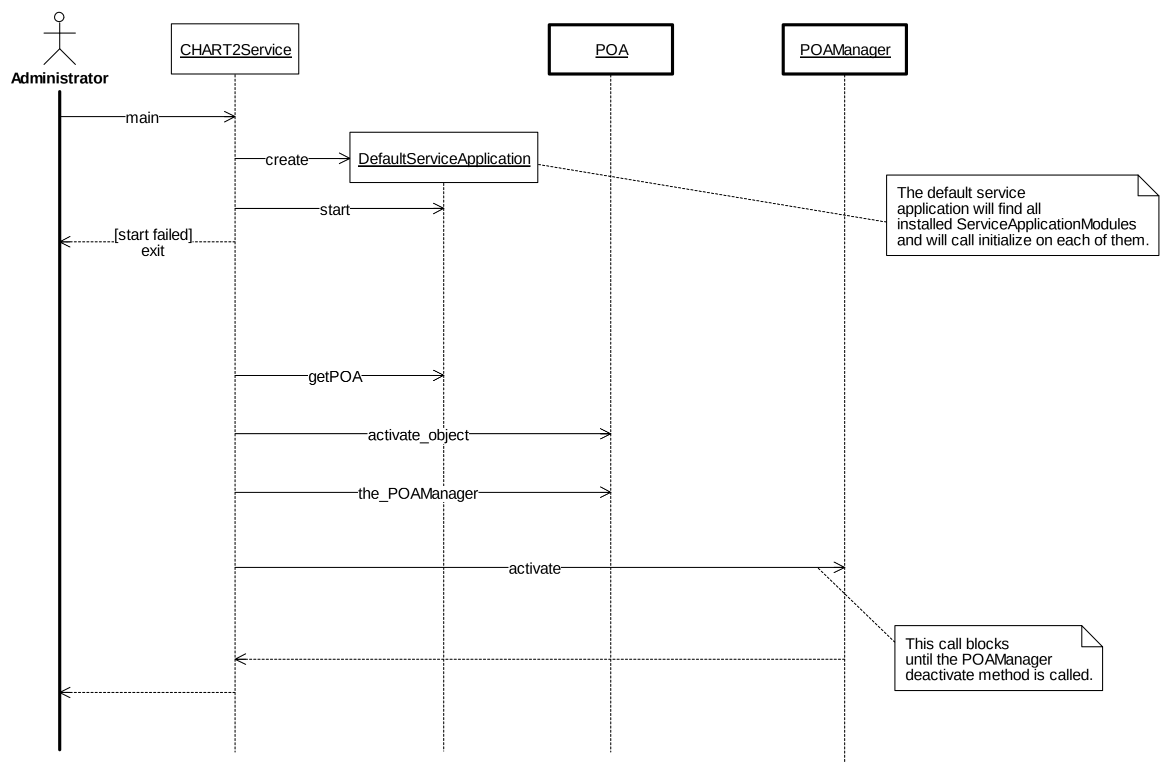

Figure 3. CHART2Service:Startup (Sequence Diagram)

3.2 CommLogModule

3.2.1 Classes

1.1.1.1 CommLogModuleClassDiagram (Class Diagram)

This Class Diagram displays classes used for managing the Communications Log. Operators can add entries directly to the Communications Log, and entries are also added indirectly with certain Traffic Events manipulations. Operators can view or search entries in the Communications Log, but cannot edit them.

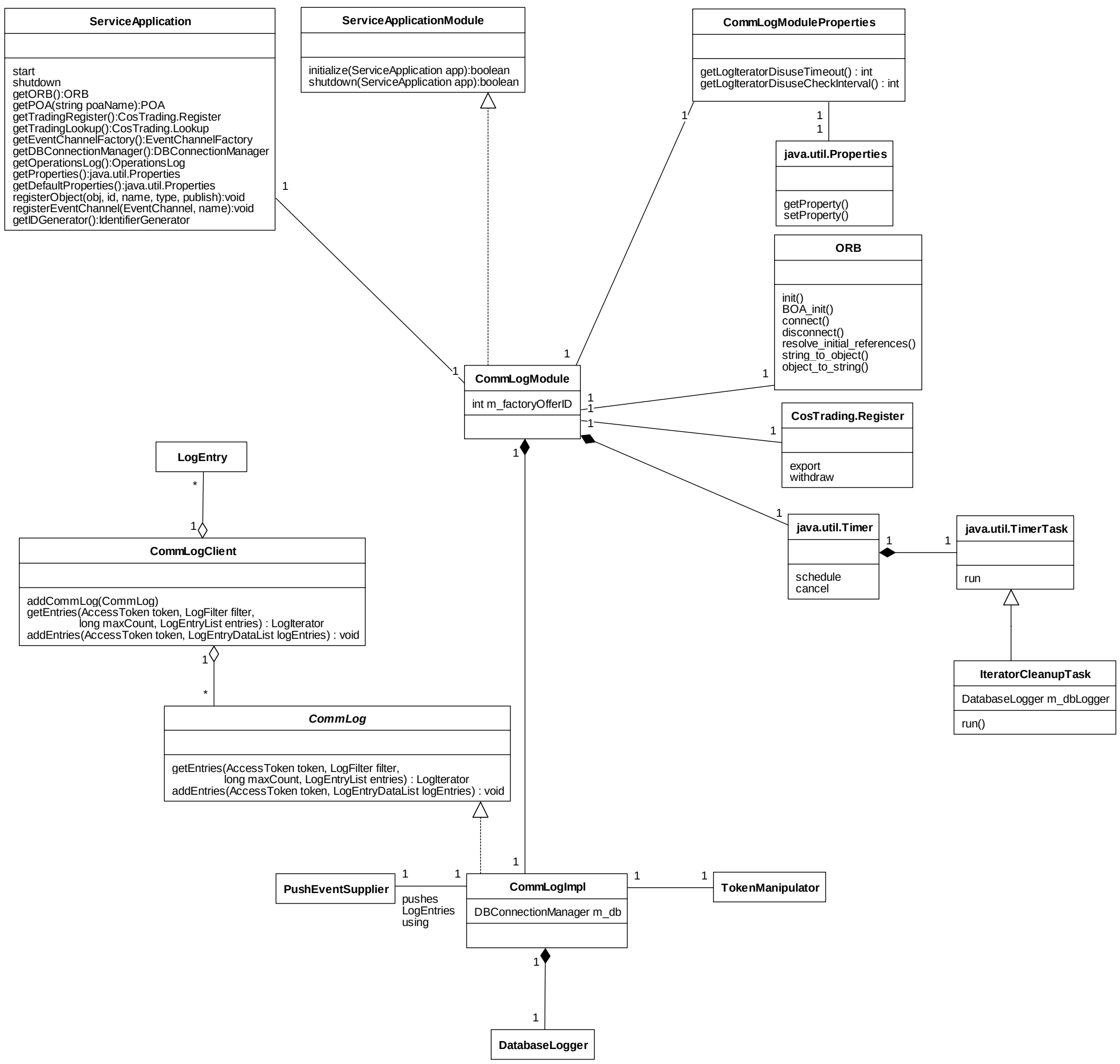

Figure 4. CommLogModuleClassDiagram (Class Diagram)

3.2.1.1.1 CommLog (Class)

This class manages log entries. These can be general Communications Log entries or specific log entries for a specific Traffic Event. This class is the primary interface for the CommLog service. It is used to persist log entries in the CHART II system and retrieve them for review. Log entries can be created directly by users or indirectly as a result of manipulating Traffic Events.

3.2.1.1.2 CommLogClient (Class)

This class is a wrapper to be used by clients of the Communications Log. It provides services such as discovering instances of the CommLog in the trader and caching entries to the comm log that are added when the comm log is not available.

3.2.1.1.3 CommLogImpl (Class)

This class implements the CommsLog interface; that is, it implements the methods defined by CommLog, allowing user interface processes access to the Communications Log for adding entries and selecting entries for viewing.

3.2.1.1.4 CommLogModule (Class)

This class implements the ServiceApplicationModule for controlling the CommLog. This class starts up the CommsLog service, and shuts it down when requested.

3.2.1.1.5 CommLogModuleProperties (Class)

This class represents an object that provides access to properties that are specific to the CommLog module.

3.2.1.1.6 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

3.2.1.1.7 DatabaseLogger (Class)

This class represents a generic database logger that can be used to log and retrieve information from the database. This class also provides a mechanism for the user to filter and retrieve logs that meet specific criteria.

3.2.1.1.8 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its “defaults”; this second property list is searched if the property key is not found in the original property list.

3.2.1.1.9 LogEntry (Class)

This class represents a typical log entry that is stored in the database. This can be a general Communications Log entry or it can be a historical entry for a Traffic Event. Some Traffic Event actions (opening, closing, etc.) are logged in the Communications Log as well as in the history of the specific Traffic Event.

3.2.1.1.10 ORB (Class)

The CORBA ORB (Object Request Broker) provides a common object oriented, remote procedure call mechanism for inter-process communication. The ORB is the basic mechanism by which client applications send requests to server applications and receive responses to those requests from servers.

3.2.1.1.11 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier’s push rate.

3.2.1.1.12 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.2.1.1.13 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.2.1.1.14 TokenManipulator (Class)

This class contains all functionality required for user rights in the system. It is the only code in the system that knows how to create, modify and check a user’s functional rights. It encapsulates the contents of an octet sequence that will be passed to every secure method. Secure methods should call the checkAccess method to validate the user. Client processes should use the check access method to verify access and optimize to reduce reduce the size of the sequence to only those rights that are necessary to invoke the secure method. The token contains the following information. Token version, Token ID, Token Time Stamp, Username, Op Center ID, Op Center IOR, functional rights

3.2.2 Sequence Diagrams

3.2.2.1 CommLogModule:addEntries (Sequence Diagram)

This sequence is initiated by a process (GUI) that is adding one or more entries into the Communications Log. (A process normally adds entries one at a time as events are created. More than one entry may be queued up if the CommsLog service has been unavailable.) The CommsLog service adds each entry on the list to the database.

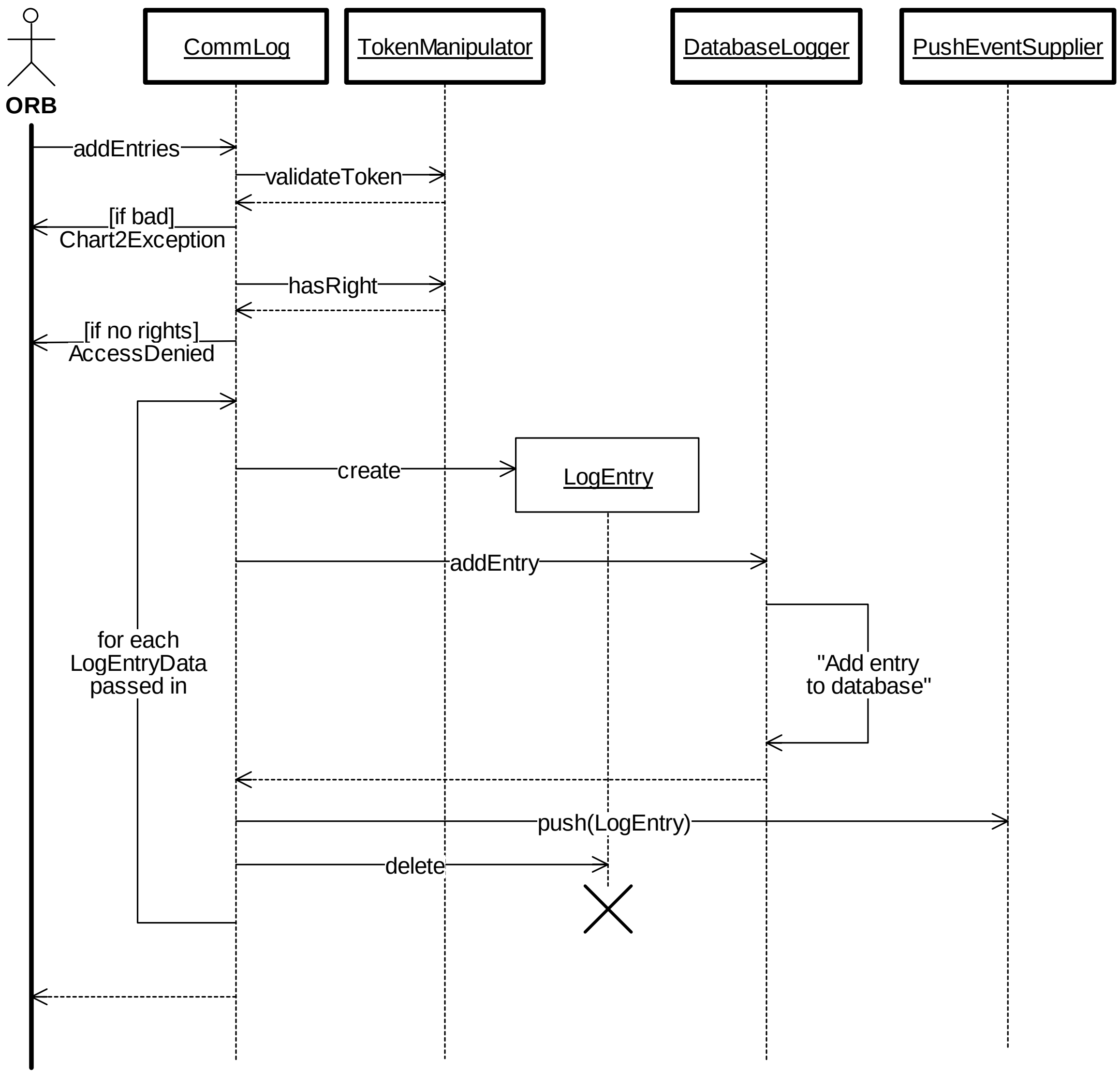

Figure 5. CommLogModule:addEntries (Sequence Diagram)

3.2.2.2 CommLogModule:destroy (Sequence Diagram)

This sequence is executed by a user process (GUI) when it is done with a LogIterator (due to no more entries left or operator cancel). Each LogEntry conceptually on the LogIterator's list which was never returned to the caller (if any) is removed from the cache and destroyed if necessary, then the LogIterator itself is deleted.

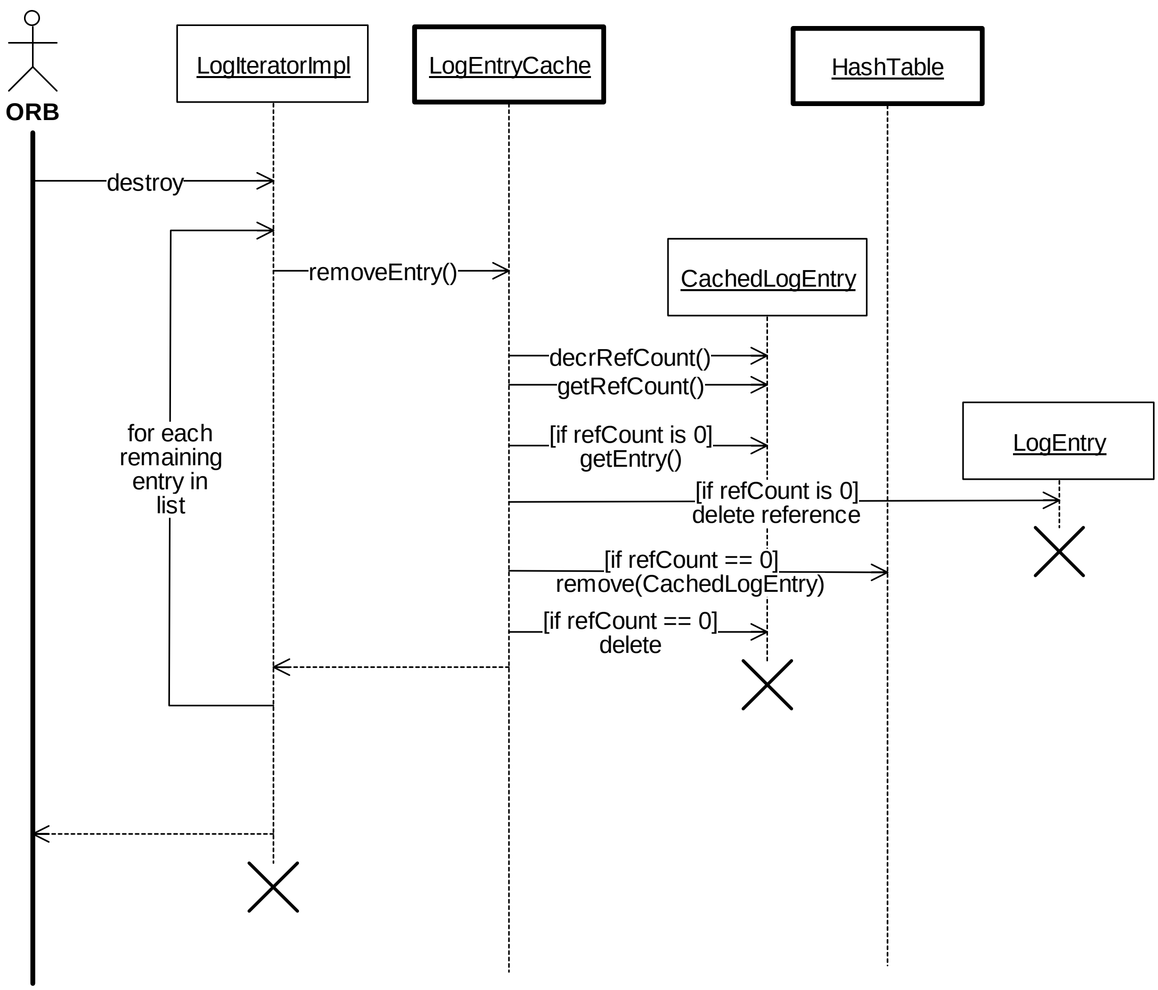

Figure 6. CommLogModule:destroy (Sequence Diagram)

3.2.2.3 CommLogModule:getEntries (Sequence Diagram)

This sequence shows how the CommsLog service responds to a request from another process (GUI) for entries from the Communications Log. The request may be constrained by a filter (based on time, originating Op Center, author, etc.). If the amount of data is larger than the requestor-specified size, the first clump is returned immediately, together with a LogIterator that can be used to later retrieve additional data, which is cached as the initial request is processed.

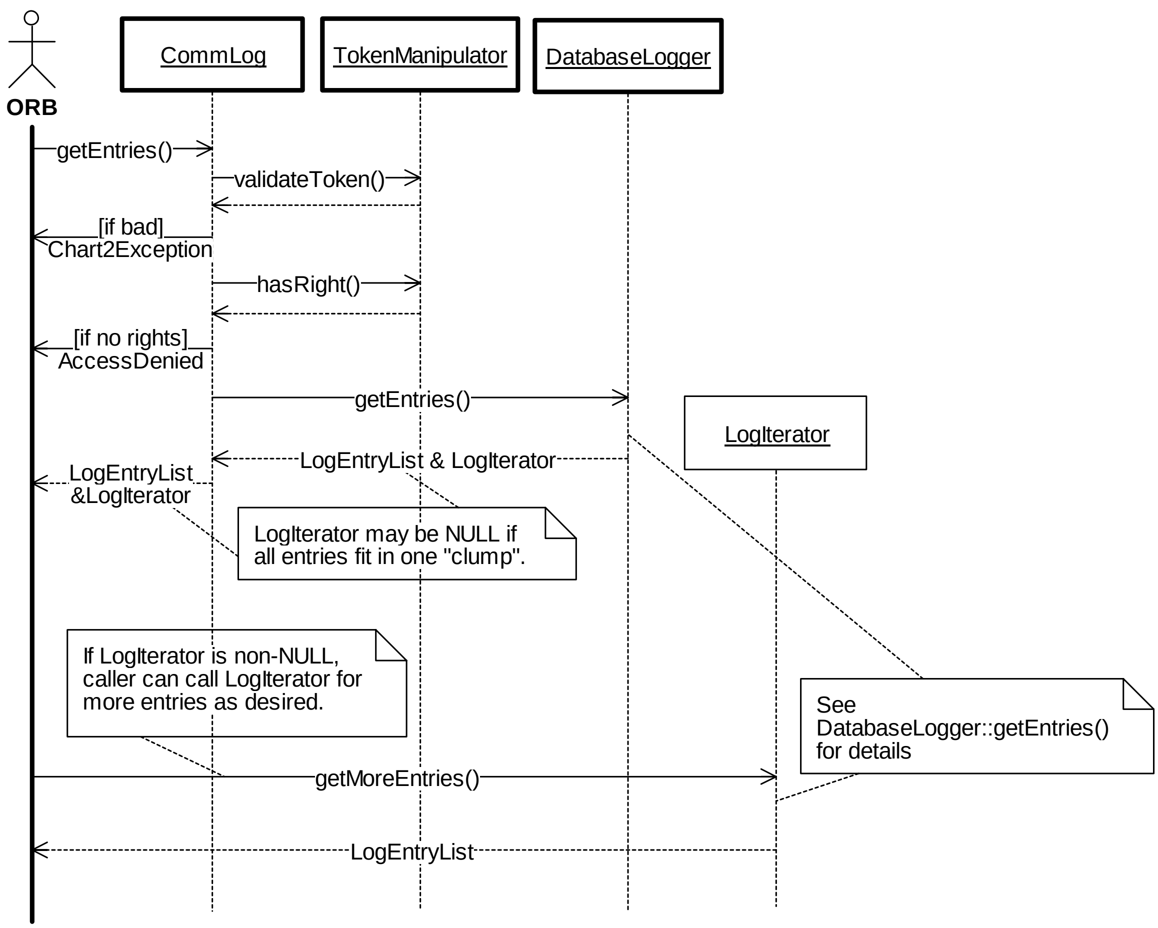

Figure 7. CommLogModule:getEntries (Sequence Diagram)

1.1.1.2 CommLogModule:initialize (Sequence Diagram)

This sequence is executed by the Service Application to start a CommsLog service if required. The CommLogModule creates a CommLog service object and makes it ready to begin servicing requests. The CommLog service allows for creation and retrieval of Communications Log Entries. New entries are pushed through the CORBA event service.

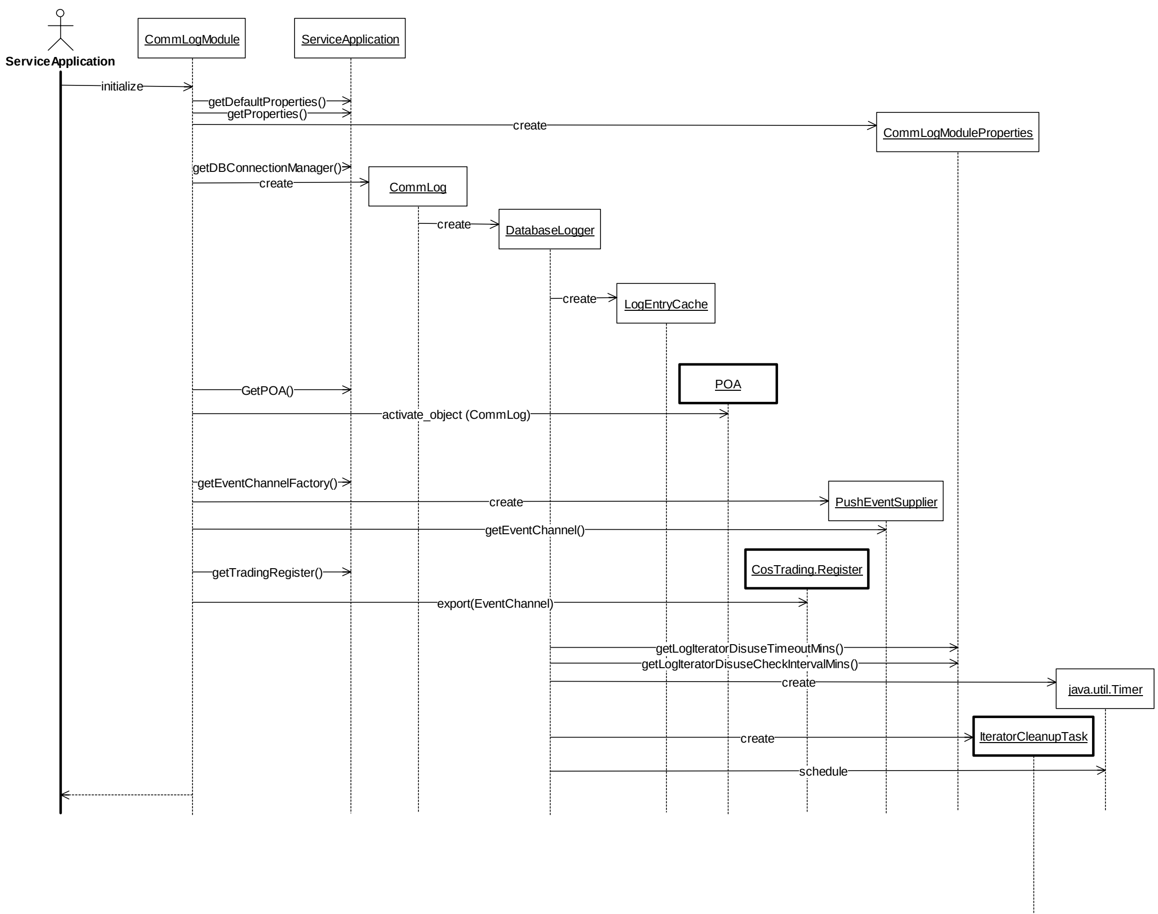

Figure 8 CommLogModule:initialize (Sequence Diagram)

1.1.1.3 CommLogModule:runIteratorCleanup (Sequence Diagram)

This sequence diagram shows the processing done to clean up any stray iterators that may have been left around by clients.

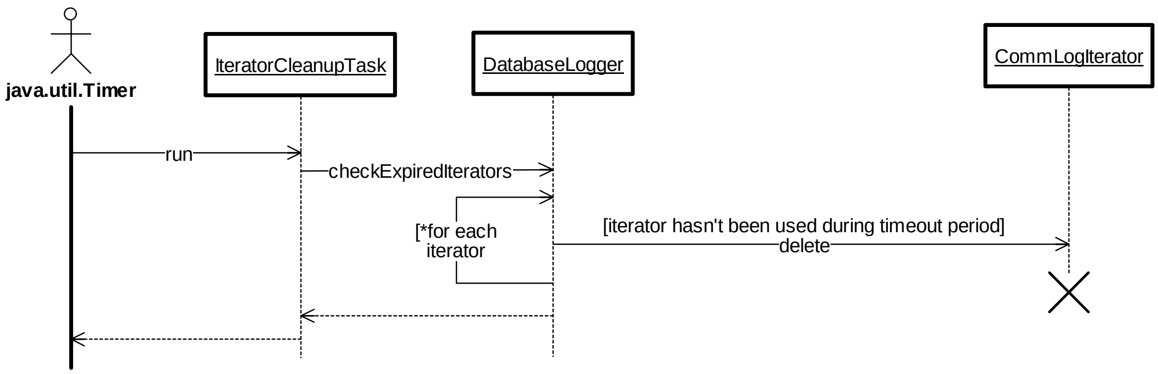

Figure 9. CommLogModule:runIteratorCleanup (Sequence Diagram)

1.1.1.4 CommLogModule:shutdown (Sequence Diagram)

This sequence is used to shutdown the CommsLog service as part of an orderly shutdown. The CommsLog deletes all memory associated with cached retrieval requests and exits. No attempt is made to persist cached data or iterators. GUIs must re-request at a later time.

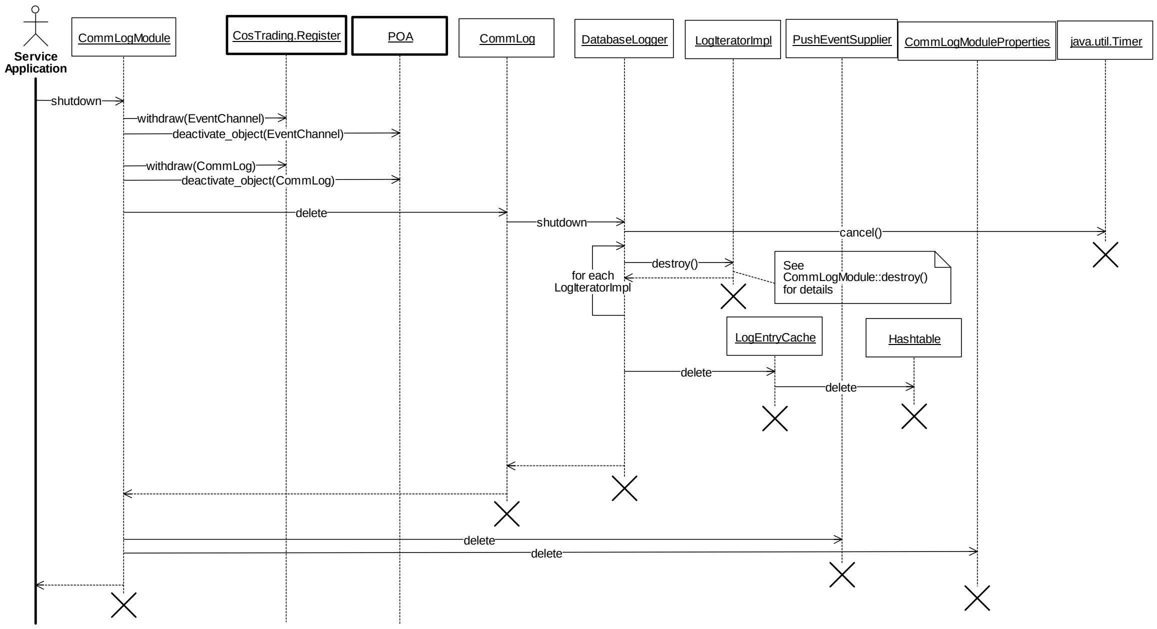

Figure 10 CommLogModule:shutdown (Sequence Diagram)

3.3 CORBAUtilities

3.3.1 Classes

3.3.1.1 CORBAClasses (Class Diagram)

The CORBAUtilities package exists to provide reference to classes that are supplied by the ORB Vendor and are referenced by other packages' class or sequence diagrams.

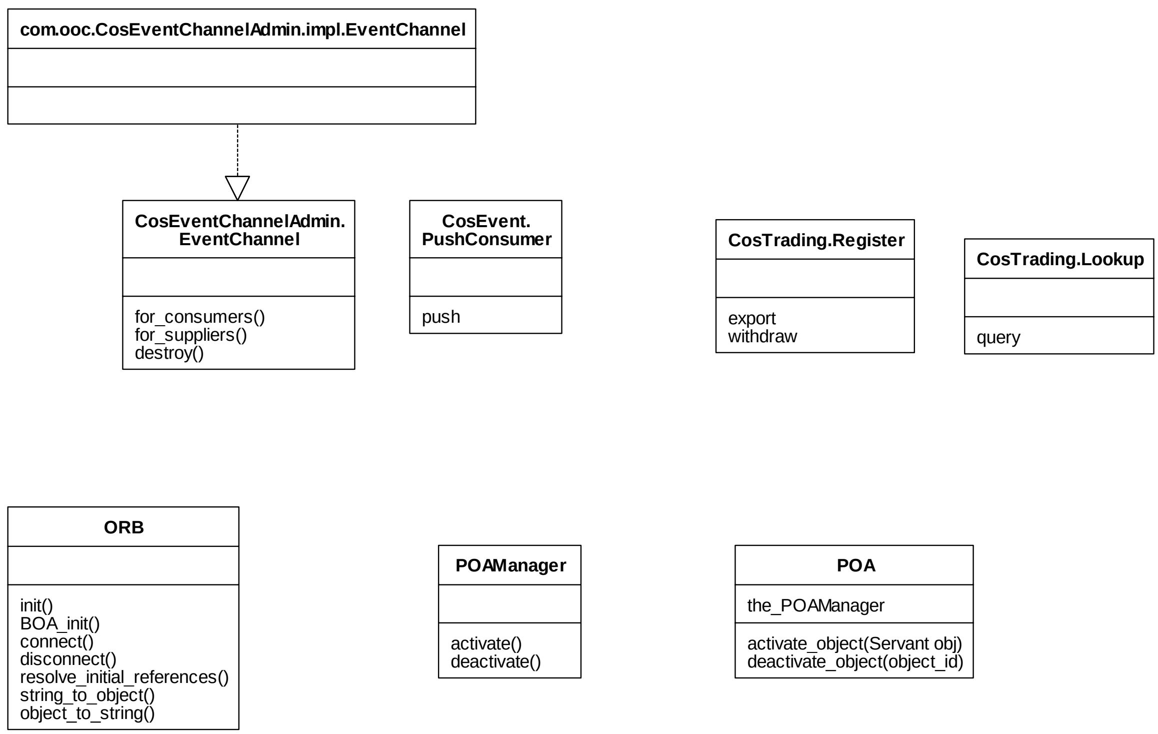

Figure 11. CORBAClasses (Class Diagram)

3.3.1.1.1 com.ooc.CosEventChannelAdmin.impl.EventChannel (Class)

This class is the ORB vendor's implementation of a CORBA event channel. The event service provided by the vendor simply serves one of these objects. The Extended Event Service serves a factory that allows multiple instances of the vendor supplied event channel to be created.

3.3.1.1.2 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

3.3.1.1.3 CosEventChannelAdmin. EventChannel (Class)

The event channel is a service that decouples the communication between suppliers and consumers of information.

3.3.1.1.4 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects that have previously been published.

3.3.1.1.5 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

3.3.1.1.6 ORB (Class)

The CORBA ORB (Object Request Broker) provides a common object oriented, remote procedure call mechanism for inter-process communication. The ORB is the basic mechanism by which client applications send requests to server applications and receive responses to those requests from servers.

3.3.1.1.7 POA (Class)

This interface represents the portable object adapter used to activate and deactivate servant objects.

3.3.1.1.8 POAManager (Class)

This interface represents the portable object adapter manager used to activate and deactivate the POA.

3.4 DeviceUtility

3.4.1 Classes

3.4.1.1 DeviceUtility (Class Diagram)

This class diagram shows utility classes that are useful for tasks in performing device control.

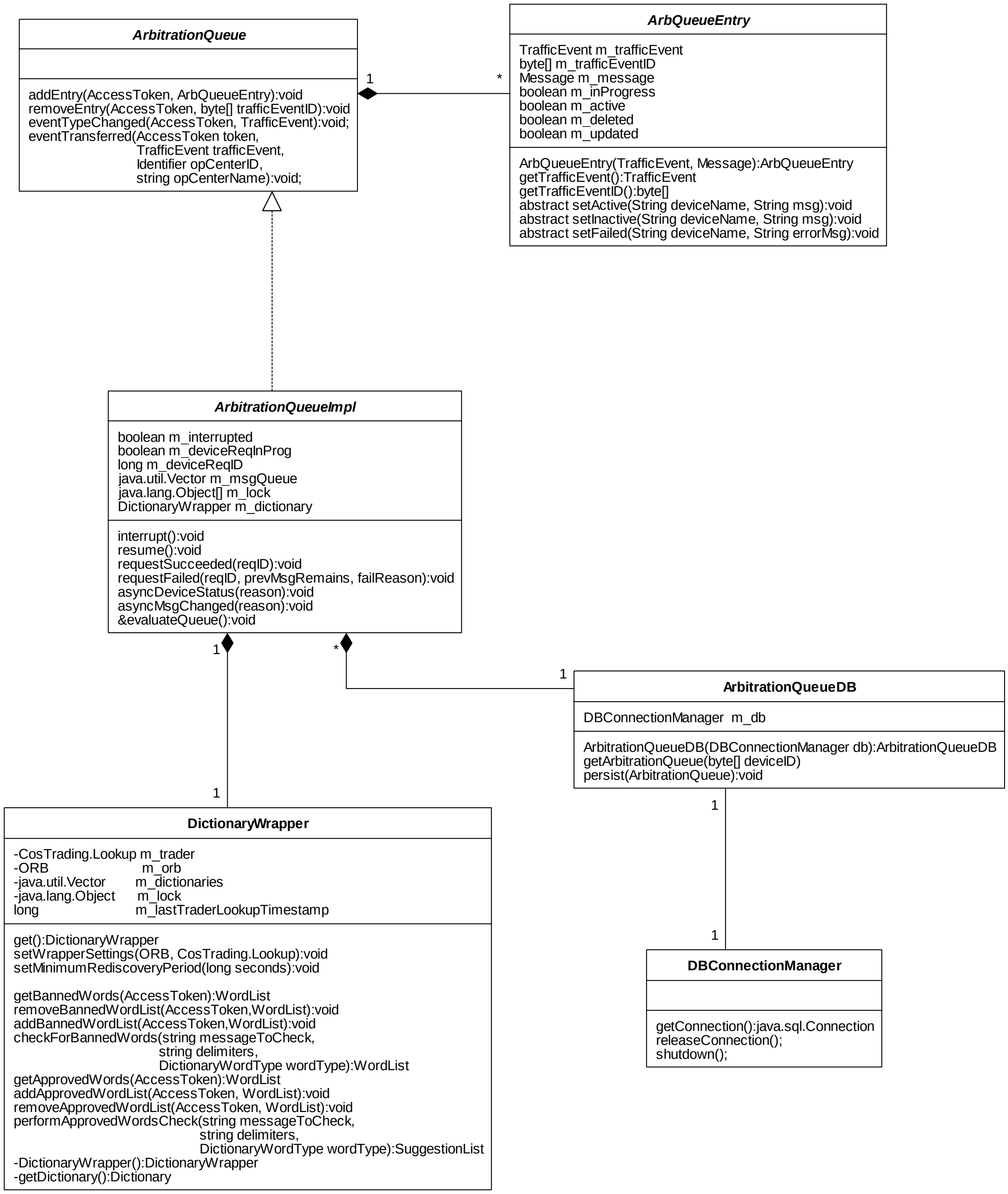

Figure 12. DeviceUtility (Class Diagram)

3.4.1.1.1 ArbitrationQueue (Class)

An ArbitrationQueue is a queue that arbitrates the usage of a device. The arbitration queue determines the proper message to be displayed on a device and switches the active message based on conditions present within the system.

For R1B2, the arbitration queue will have a single slot (and is therefore not really a queue). The device arbitration rules used by the arbitration queue in R1B2 rely on user rights and operations centers. When the arbitration queue's slot is in use, another message can only be added to the queue if the user adding the message is from the same operations center that is responsible for the existing message, or the user has a special functional right that allows this rule to be overridden.

The arbitration queue can be interrupted to keep it from performing its automated processing. This mode is used to allow maintenance on the device being arbitrated by the queue without having the queue's automatic processing interfere with the maintenance activities.

When an interrupted arbitration queue is taken out of its interrupted state through the use of the resume method, the arbitration queue evaluates the messages in the queue and restores the device to the proper state. For R1B2, the arbitration queue performs no special processing when resumed because the queue cleans itself when interrupted and does not allow new entries while interrupted.

3.4.1.1.2 ArbitrationQueueDB (Class)

This class handles the database interaction for the arbitration queue. The HAR module initializes this class with the HAR database connection. Messages added to the queue are also added to the database and removed from the database when they are removed from the queue.

3.4.1.1.3 ArbitrationQueueImpl (Class)

This class is an implementation of the ArbitrationQueue interface as defined by the IDL. This class arbitrates the usage of a messaging device (DMS or HAR) among multiple users. For R1B2, the arbitration algorithm is a "last in wins" scheme, where the last request to use the device being arbitrated overwrites any previous requests. When an arbitrated device is in use, the operations center of the requester is used to determine if the request will be allowed on the queue. Only a user from the same operations center that currently has a message on a device is allowed to overwrite a previous message. On exception to this is that users with a special functional right may override messages that were set from operations centers other than their own.

3.4.1.1.4 ArbQueueEntry (Class)

This class is used for an entry on the arbitration queue for a single message for a single traffic event / response plan item. The class holds the associated message, traffic event, and response plan item.

3.4.1.1.5 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two separate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.4.1.1.6 DictionaryWrapper (Class)

This singleton class provides a wrapper for the system dictionary that provides automatic location of the dictionary and automatic re-discovery should the dictionary reference return an error. This class also allows for built-in fault tolerence by automatically failing over to a "working" dictionary without the user of this class being aware that this being done. In addition, this class defers the discovery of the Dictionary until its first use, thus eliminating a start-up dependency for modules that use the dictionary.

This class delegates all of its method calls to the system dictionary using its currently known good reference to the system dictionary. If the current reference returns a CORBA failure in the delegated call, this class automatically switches to another reference. When there are no good references (as is true the first time the object is used), this class issues a trader query to (re)discover the published Dictionary objects in the system. During a method call, the trader will be queried at most one time and under normal circumstances (other than the first use) the trader will not be queried at all.

3.4.2 Sequence Diagrams

3.4.2.1 ArbQueueProcessing:addEntry (Sequence Diagram)

This diagram shows the processing involved when an entry is added to an arbitration queue. The arbitration queue blocks the addition of the entry if the user does not have the proper functional rights or the top entry on the queue is owned by an operations center other than the requestor's and the requestor does not have override rights. If the prior checks succeed, the entry is added to the head of the message queue and any prior entries that are not in progress or active are notified that they will not be placed on the device. If the queue does not already have a request to set a message on the device in progress, the abstract evaluate queue method is called and it performs processing as implemented by the derived class.

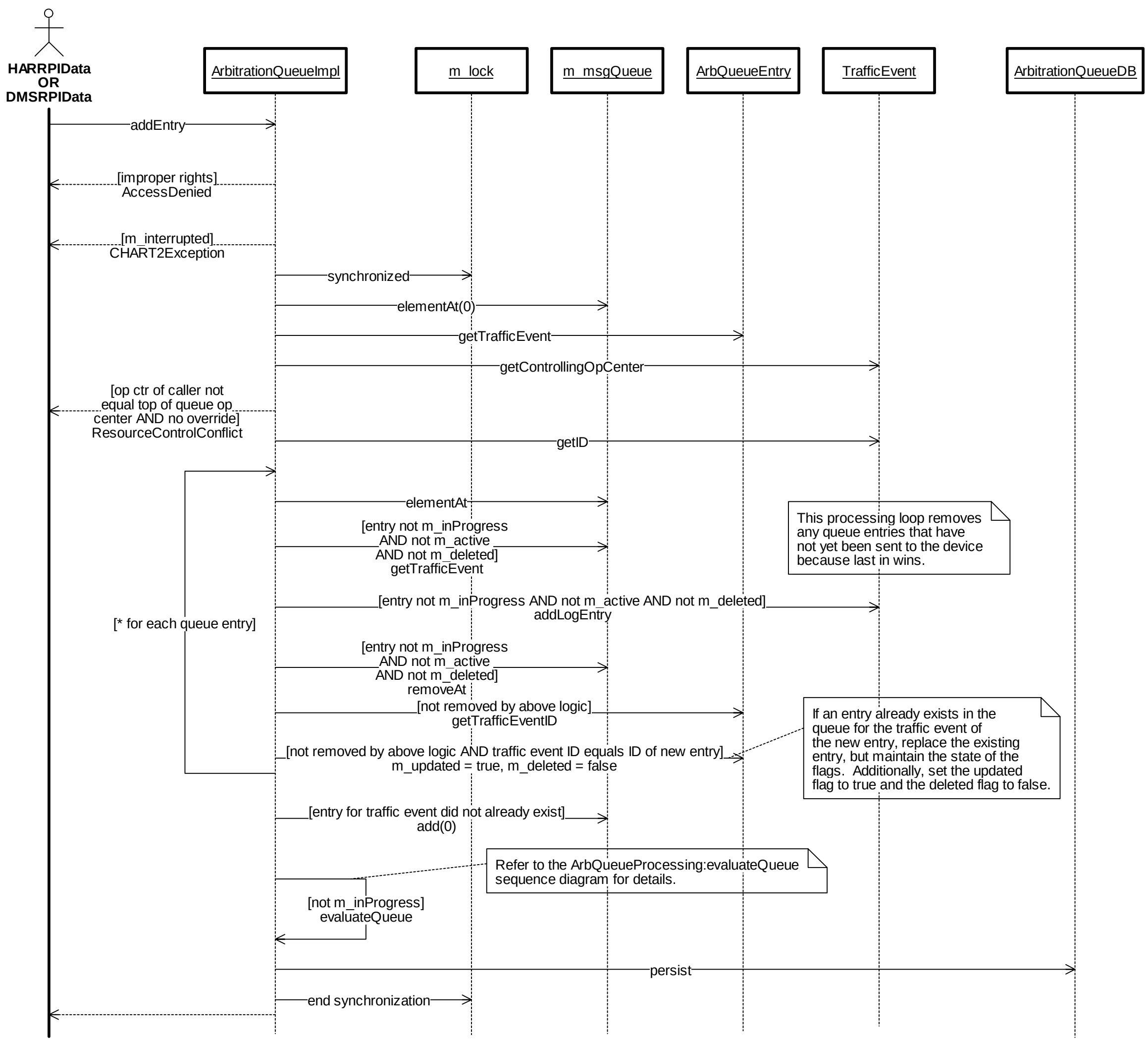

Figure 13. ArbQueueProcessing:addEntry (Sequence Diagram)

3.4.2.2 ArbQueueProcessing:asyncMsgChanged (Sequence Diagram)

This diagram shows the processing that occurs when a device detects that its message has been changed and it notifies the arbitration queue of this condition. This typically only applies to a polled device, such as a DMS, which may detect a comm failure and then mark the device blank after the device is comm failed for a pre-determined length of time. When notified of this condition, the arbitration queue notifies all entries that are currently active that they are no longer active and removes them from the queue.

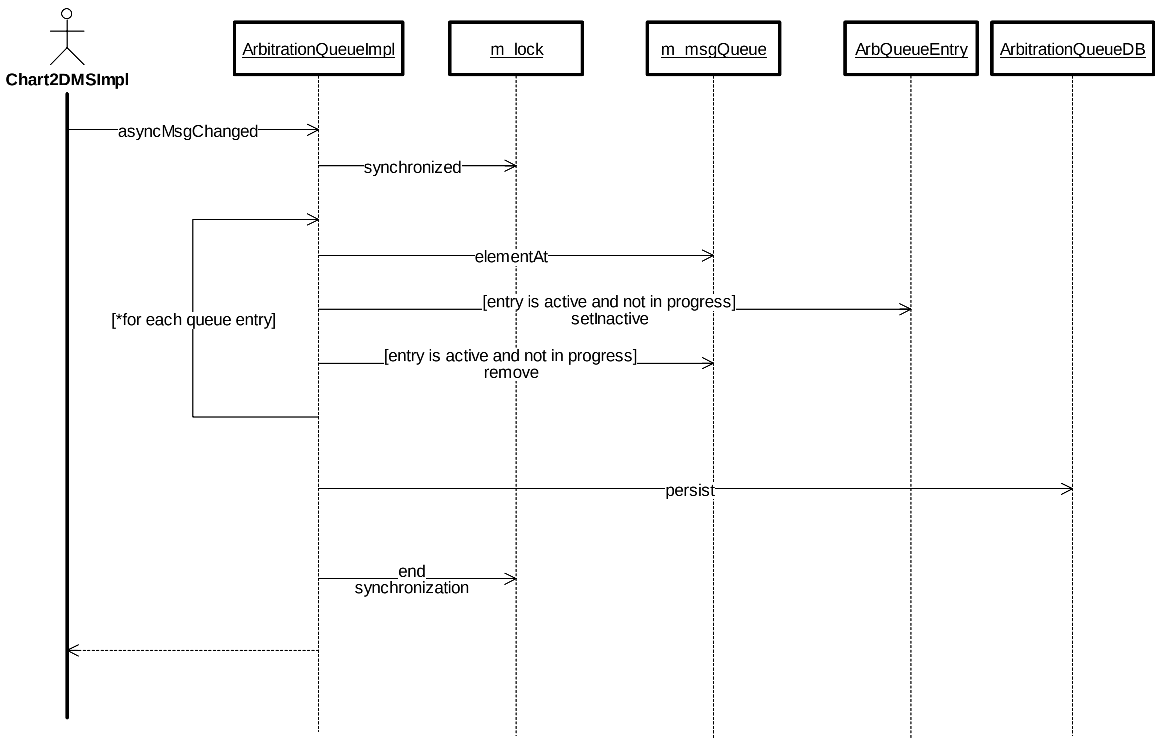

Figure 14. ArbQueueProcessing:asyncMsgChanged (Sequence Diagram)

3.4.2.3 ArbQueueProcessing:evaluateQueue (Sequence Diagram)

This diagram shows the processing of the ArbitrationQueue's evaluateQueue method, which is abstract and must be implemented by derived classes. The processing done for derived classes is similar except for the type of device type (and method signature) that is called to set a message on the device or blank the device. This method decides what action to take based on the entries on the queue. If the top entry on the queue is not marked for deletion and is not active, a request is issued to the device to set the message on the device. If all remaining entries on the queue are marked for deletion (only one possible for R1B2), a request is sent to blank the device. After the device has processed a request originated from the arbitration queue, it calls one of the requestSucceeded or requestFailed methods, at which time the queue performs houskeeping. Refer to the ArbQueueProcessing:requestSucceeded, ArbQueueProcessing:requestFailed for more details.

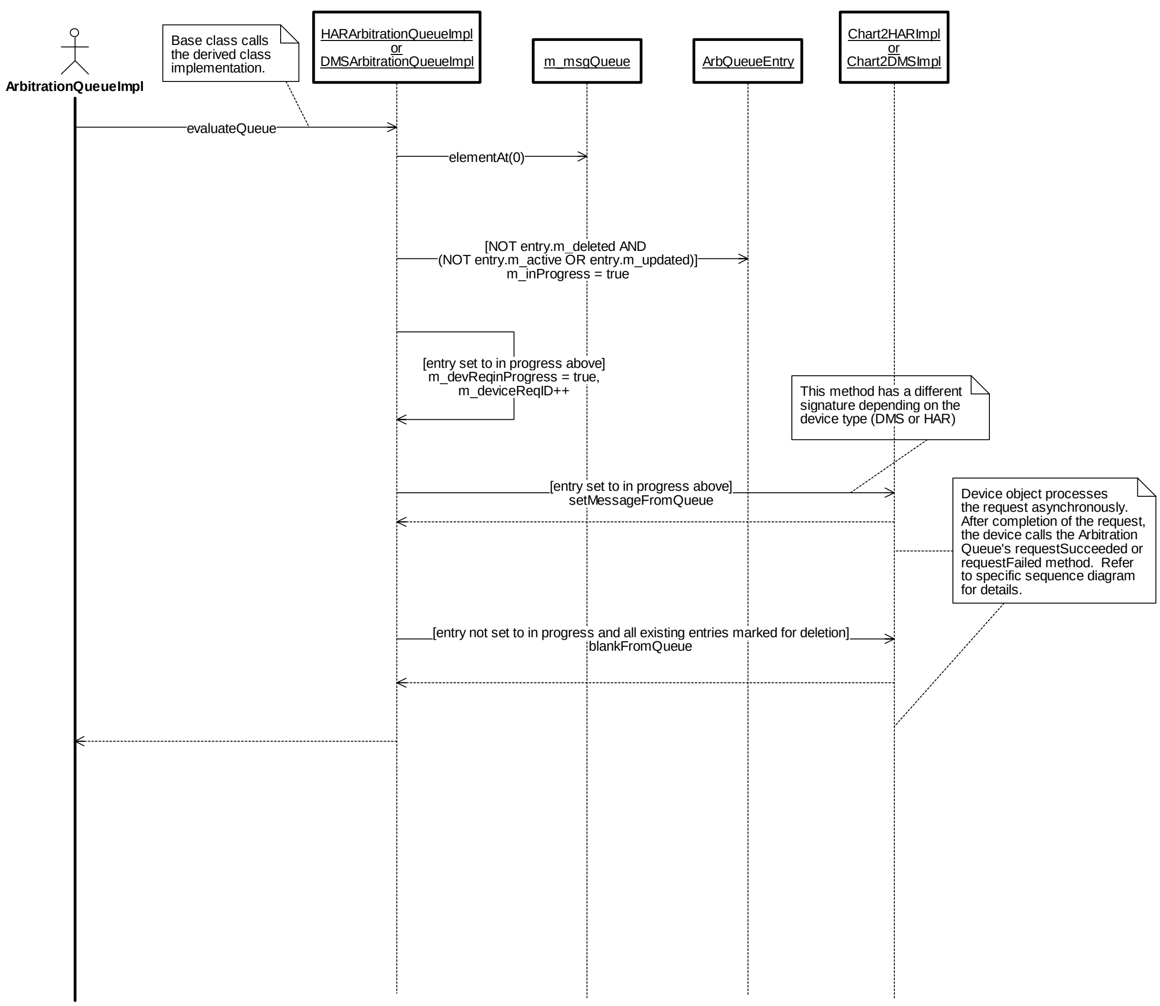

Figure 15. ArbQueueProcessing:evaluateQueue (Sequence Diagram)

3.4.2.4 ArbQueueProcessing:interrupt (Sequence Diagram)

This diagram shows the processing that occurs when the arbitration queue is interrupted. The arbitrated device interrupts the arbitration queue when the device is taken offline or put in maintenance mode to keep the arbitration queue from attempting to put messages on the device. In R1B2, messages on the arbitration queue are not re-activated so when it is interrupted it removes each entry and notifies it that it is no longer active.

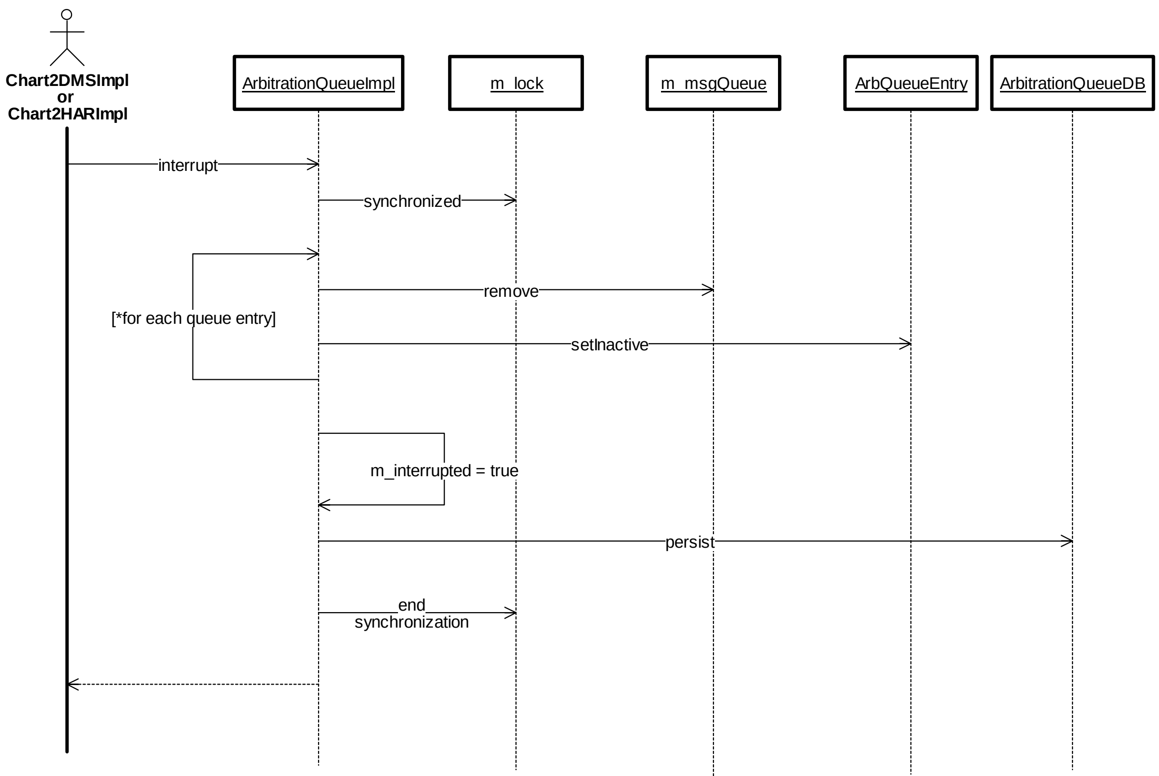

Figure 16. ArbQueueProcessing:interrupt (Sequence Diagram)

3.4.2.5 ArbQueueProcessing:removeEntry (Sequence Diagram)

This diagram shows the processing involved when an entry is removed from the arbitration queue. The ID of the traffic event to be removed is used to find the corresponding queue entry and the entry is marked for deletion. If an arbitration queue request is in progress, any action regarding the deletion is deferred until after the current request is completed. If no request is in progress and the entry being deleted is not active, the entry is removed from the queue and its traffic event is notified. The abstract evaluateQueue method is then called which may decide to replace the active message or blank the device.

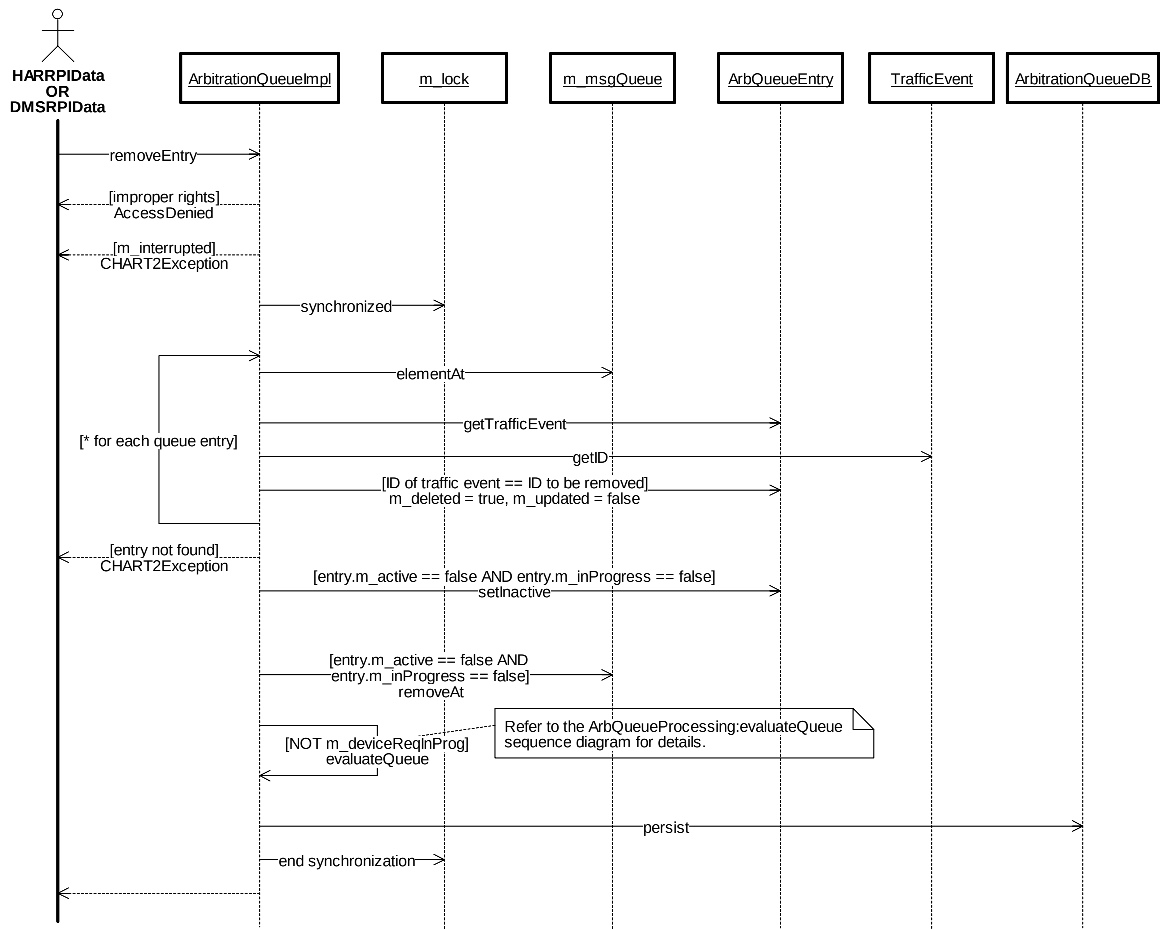

Figure 17. ArbQueueProcessing:removeEntry (Sequence Diagram)

3.4.2.6 ArbQueueProcessing:requestFailed (Sequence Diagram)

This diagram shows the processing that occurs when an arbitrated device completes a request from the arbitration queue and the request has failed. When this occurs, the device calls the arbitration queue's requestFailed method and indicates if the failure affected the previous message that was on the sign. The arbitration queue performs some house keeping on its queue entries, notifying the owner of the message that was being activated of the failure, and deactivating all other entries if the message on the device is not able to be determined due to the type of failure. Inactive entries are removed from the queue for in R1B2 messages are not kept automatically re-activated.

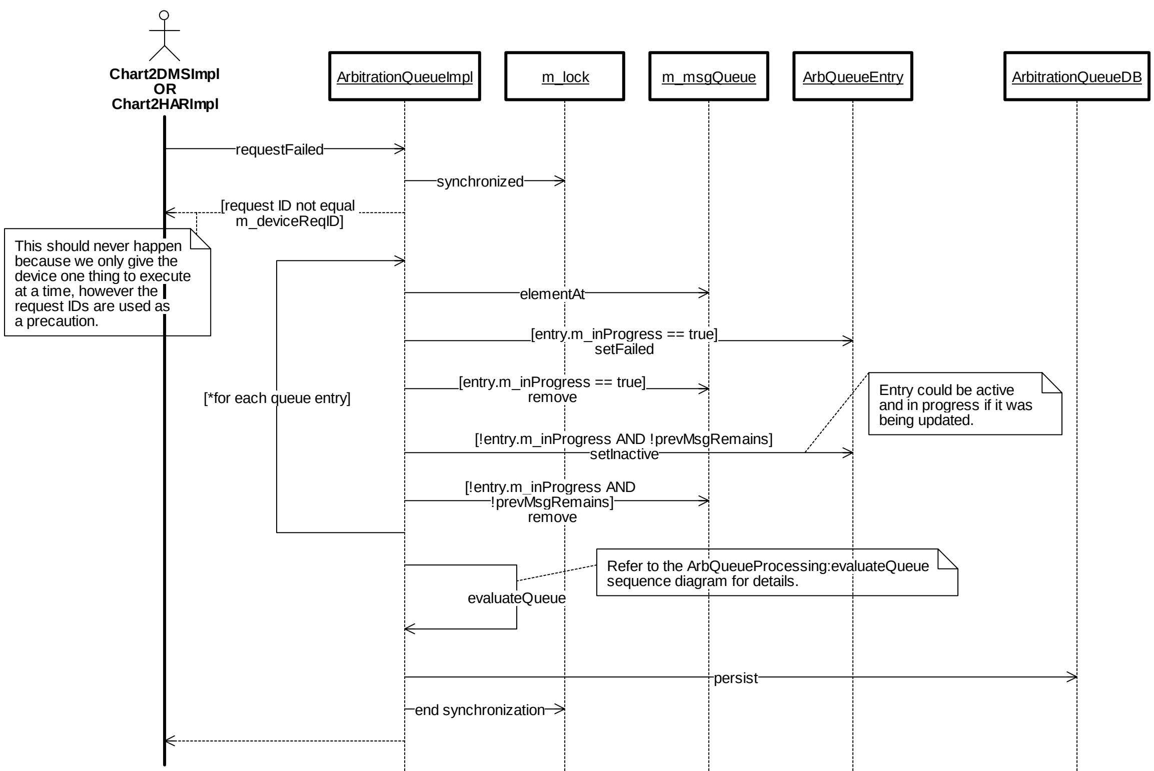

Figure 18. ArbQueueProcessing:requestFailed (Sequence Diagram)

3.4.2.7 ArbQueueProcessing:requestSucceeded (Sequence Diagram)

This diagram shows the processing that occurs when an arbitrated device completes a request from the arbitration queue and notifies the arbitration queue that the request succeeded. When this occurs, the arbitration queue does housekeeping on its queue entries. Any entries that were previously marked as active are notified that they are inactive and are removed from the queue. Any entries that were previously marked as in progress are marked as active and are notified that they are active. When an entry's setActive or setInactive method is called, a log entry is made in the traffic event and the response plan item that added the entry to the queue is notified that it is no longer active.

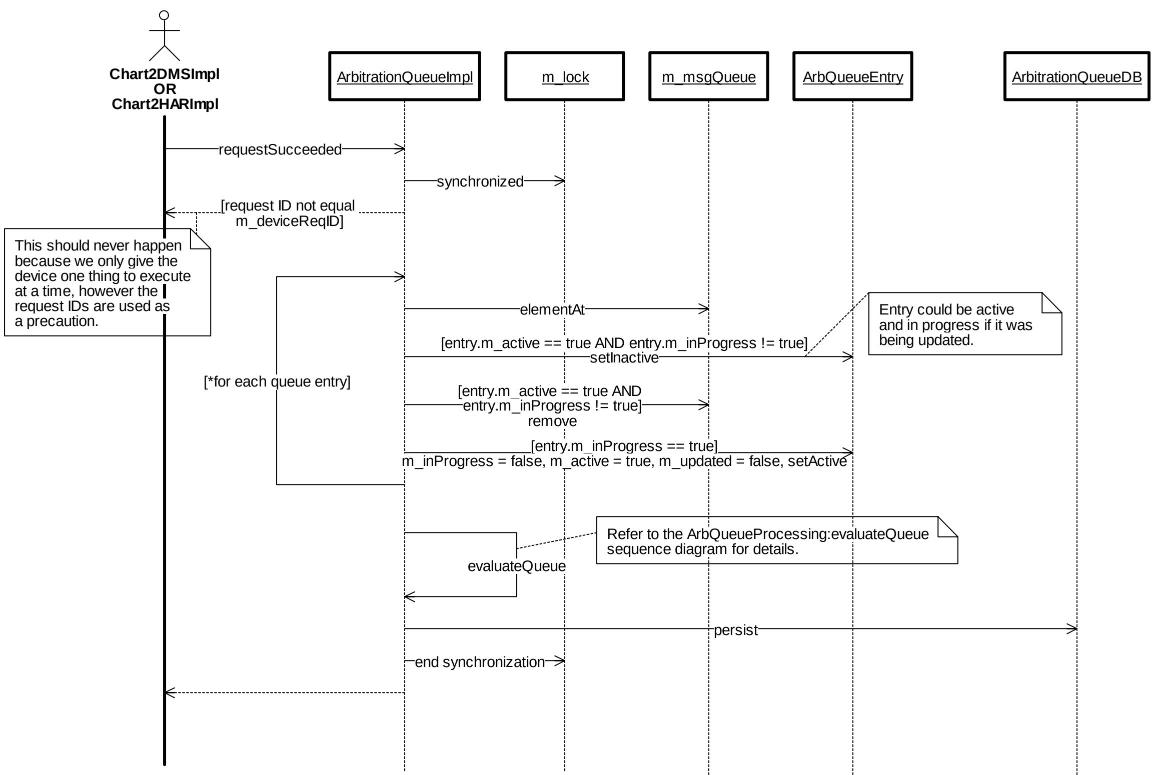

Figure 19. ArbQueueProcessing:requestSucceeded (Sequence Diagram)

3.4.2.8 ArbQueueProcessing:resume (Sequence Diagram)

This diagram shows the processing that occurs when the arbitration queue is told to resume its processing. In R1B2, because the queue is emptied when it is interrupted, the only processing that takes place is to set an internal flag and return.

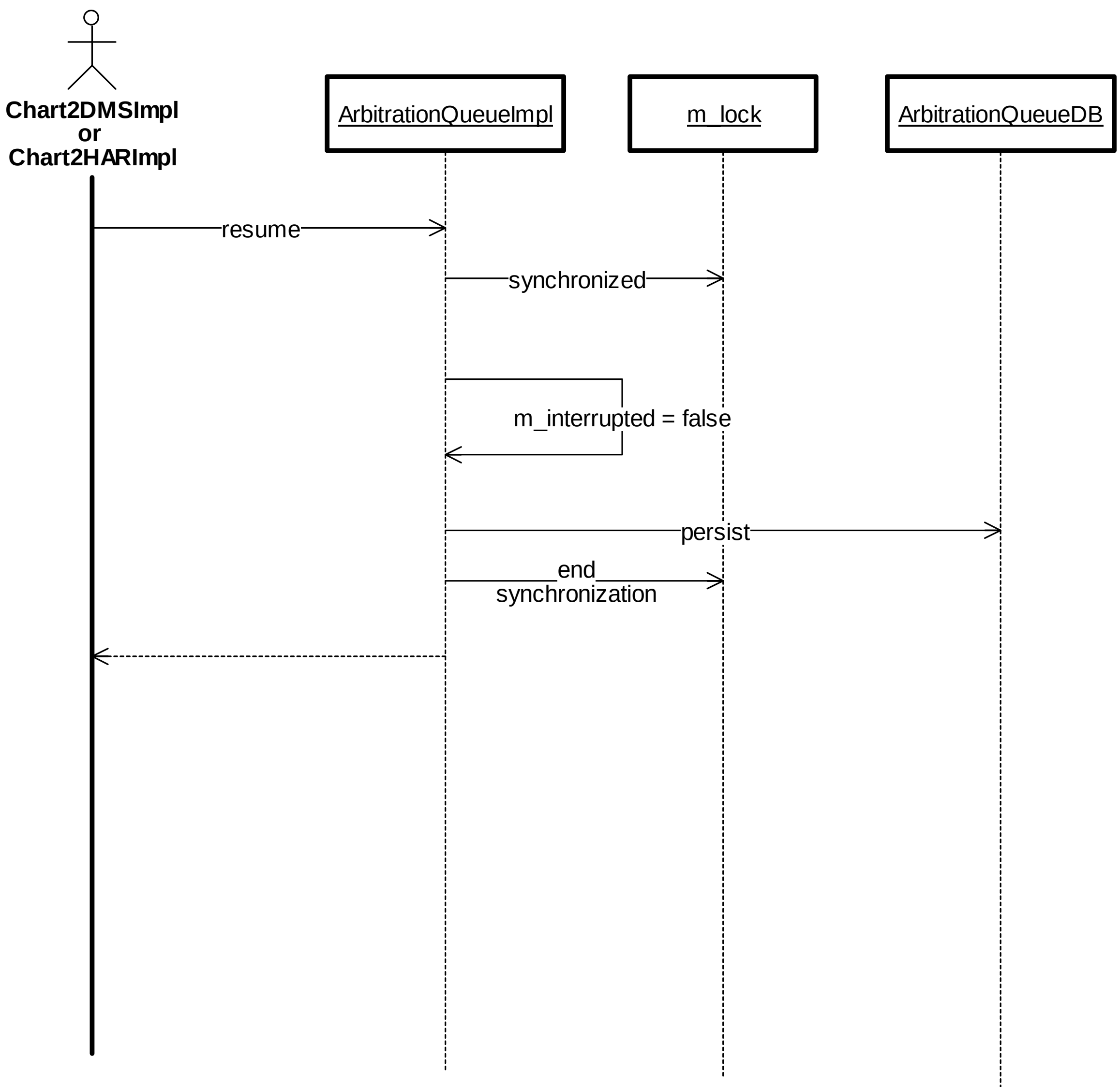

Figure 20. ArbQueueProcessing:resume (Sequence Diagram)

3.5 DictionaryModule

3.5.1 Classes

3.5.1.1 DictionaryModClassDiagram (Class Diagram)

The DictionaryModule is a Service Application module that creates and serves the Dictionary implementation to the rest of the CHART2 system.

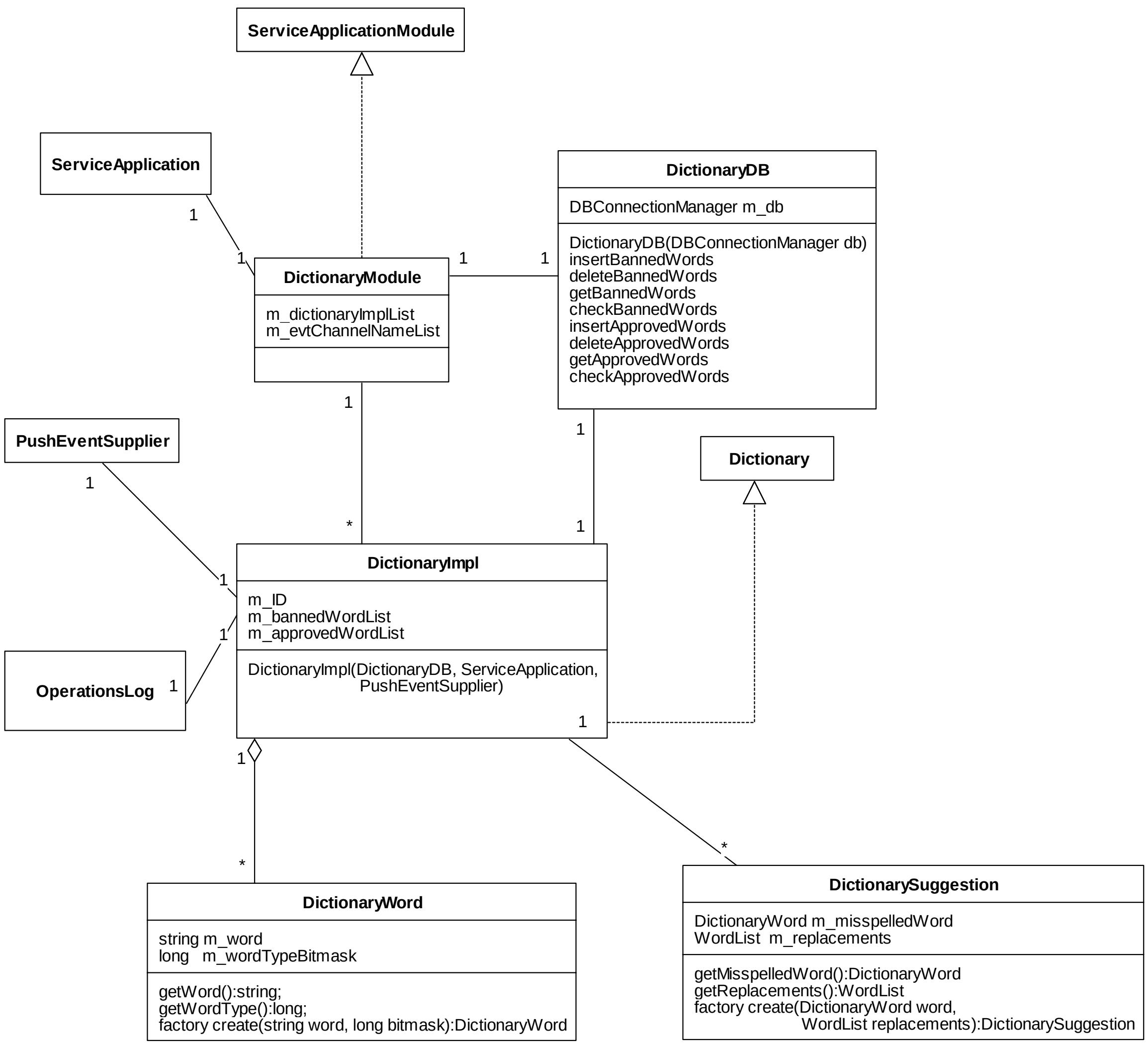

Figure 21. DictionaryModClassDiagram (Class Diagram)

3.5.1.1.1 Dictionary (Class)

The Dictionary IDL interface provides functionality to add, delete and check for words that are approved or banned from being used in a CHART2 messaging device. Examples of messaging devices are DMS, HAR, etc.

3.5.1.1.2 DictionaryDB (Class)

This class provides API calls to add, remove and retrieve banned words and approved words from the database. The connection to the database is acquired from the Database object that manages all the database connections.

3.5.1.1.3 DictionaryImpl (Class)

This class implements the Dictionary as specified by the IDL. It provides functionality to add, delete and check for words that are banned or approved from being used in a DMS message.

3.5.1.1.4 DictionaryModule (Class)

This class implements the Service Application module interface. It publishes the dictionary implementation.

3.5.1.1.5 DictionarySuggestion (Class)

A DictionarySuggestion represents a list of suggested words that may be used as a substitute for the word that could not be found in the approved words dictionary database.

3.5.1.1.6 DictionaryWord (Class)

A DictionaryWord represents a word in the chart2 dictionary. It contains information that qualifies the type of devices that the word applies to.

3.5.1.1.7 OperationsLog (Class)

This class provides the functionality to add a log entry to the CHART II operations log. At the time of instantiation of this class, it creates a queue for log entries. When a user of this class provides a message to be logged, it creates a time-stamped OpLogMessage object and adds this object to the OpLogQueue. Once queued, the messages are written to the database by the queue driver thread in the order they were queued.

3.5.1.1.8 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier's push rate.

3.5.1.1.9 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.5.1.1.10 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.5.2 Sequence Diagrams

3.5.2.1 DictionaryModule:initialize (Sequence Diagram)

When the DMS service calls the initialize method of Dictionary module, the dictionary objects are created, connected to the ORB, exported to the CORBA trading service. The dictionary objects are now available to serve the consumers.

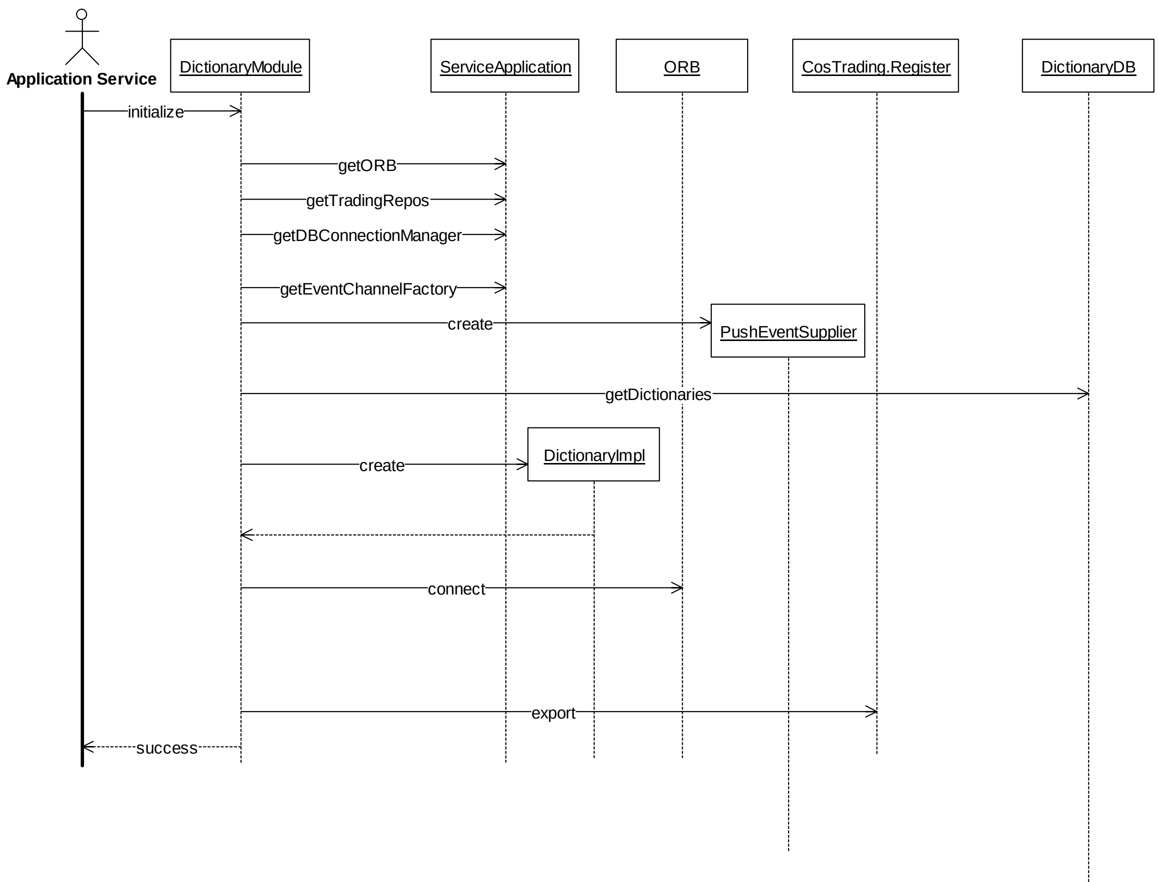

Figure 22. DictionaryModule:initialize (Sequence Diagram)

3.5.2.2 DictionaryModule:shutdown (Sequence Diagram)

When the host service application calls shutdown in the Dictionary module, the dictionary object is withdrawn from the CORBA trading service and disconnected from the ORB. The objects are then deleted.

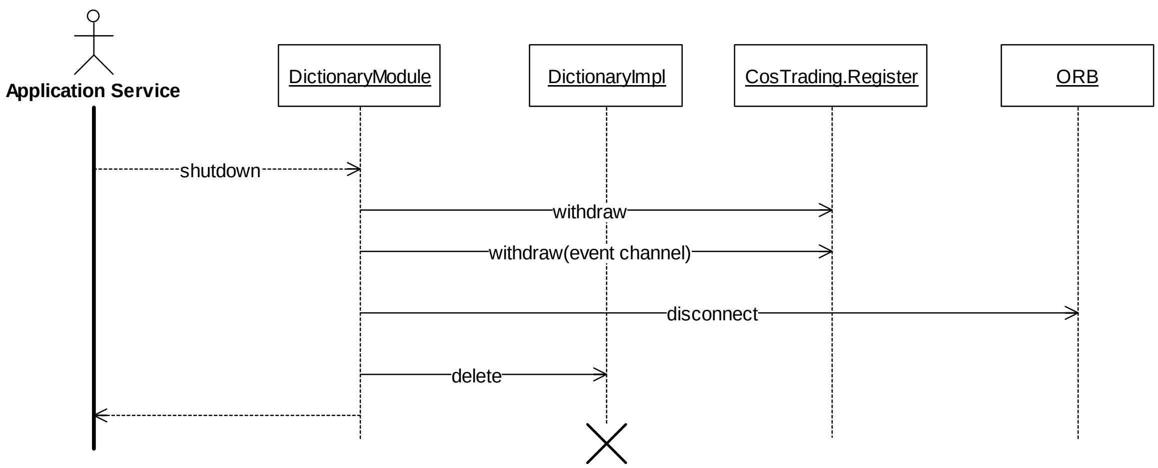

Figure 23. DictionaryModule:shutdown (Sequence Diagram)

3.5.2.3 DictionaryImpl:addApprovedWordList (Sequence Diagram)

The given list of words is added to the approved words dictionary database. The newly added words are then communicated to the dictionary event consumers by invoking the push operation. Access is denied to any operator without the “Manage Dictionary” privilege.

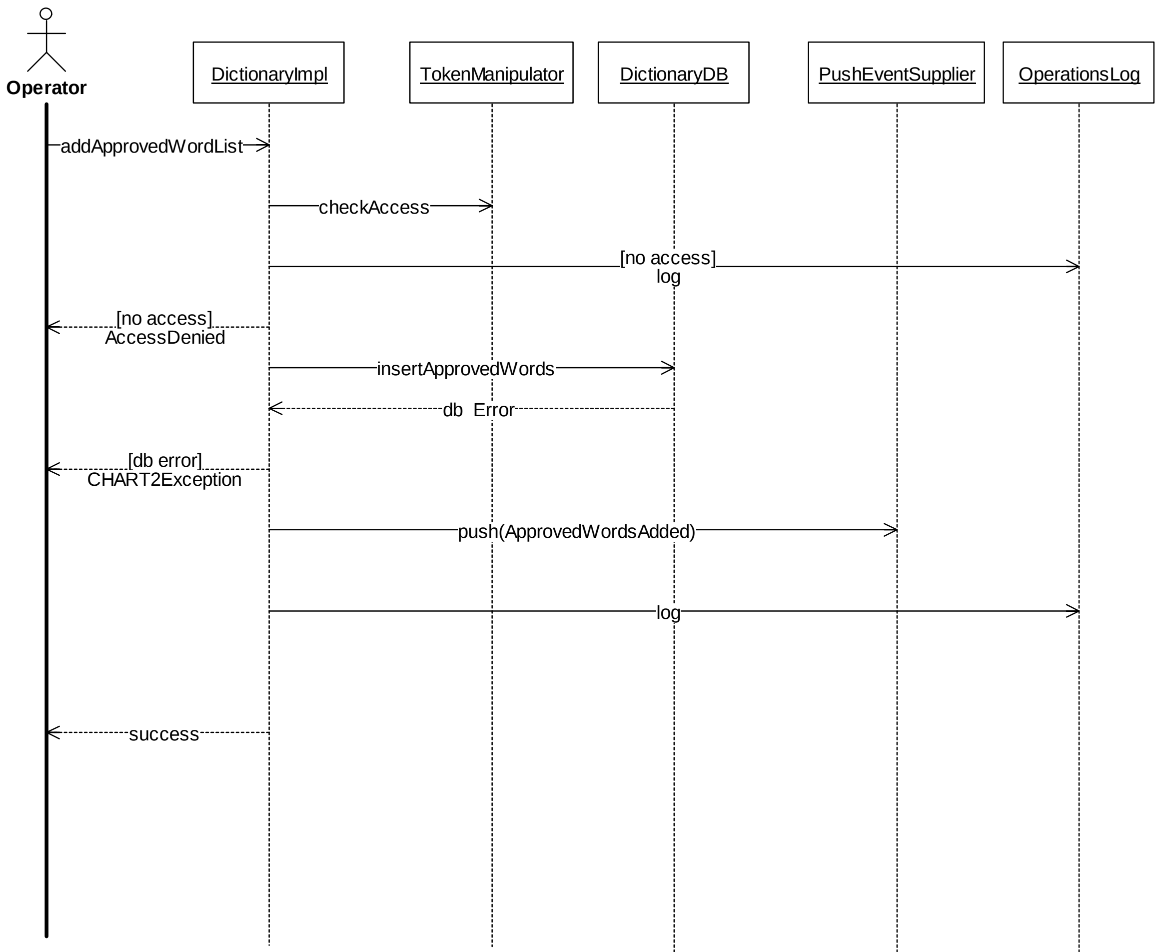

Figure 24. DictionaryImpl:addApprovedWordList (Sequence Diagram)

3.5.2.4 DictionaryImpl:addBannedWordList (Sequence Diagram)

The given list of words is added to the banned words dictionary database and the copy of the dictionary in memory is also updated. The newly added banned words are then communicated to the dictionary event consumers by invoking the push operation. Access is denied to any operator without the “Manage Dictionary” privilege.

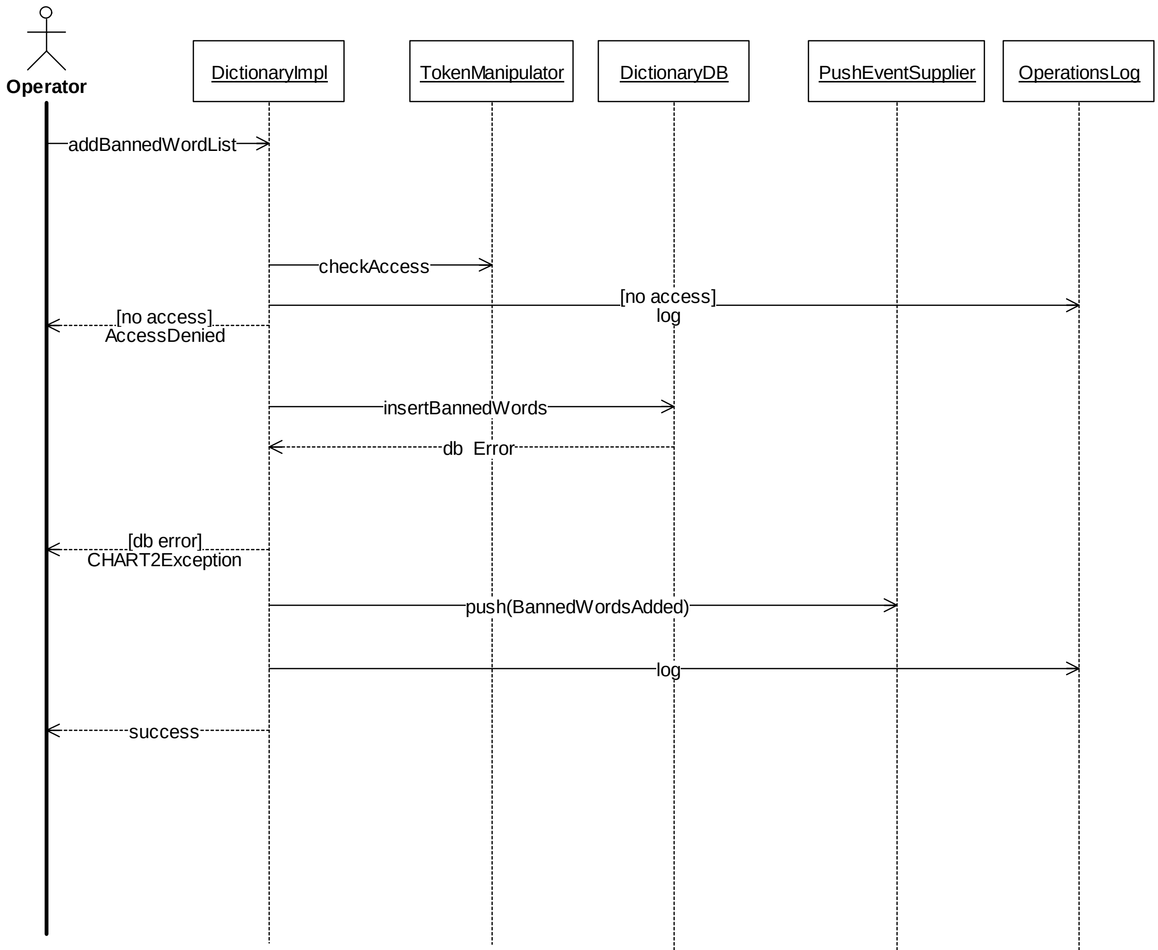

Figure 25. DictionaryImpl:addBannedWordList (Sequence Diagram)

3.5.2.5 DictionaryImpl:checkForBannedWords (Sequence Diagram)

The string provided by the operator is scanned for any banned words by looking up the database. Any character from the given set of delimiters is taken to be a valid delimiter of words in the string. The list of banned words present in the string is returned.

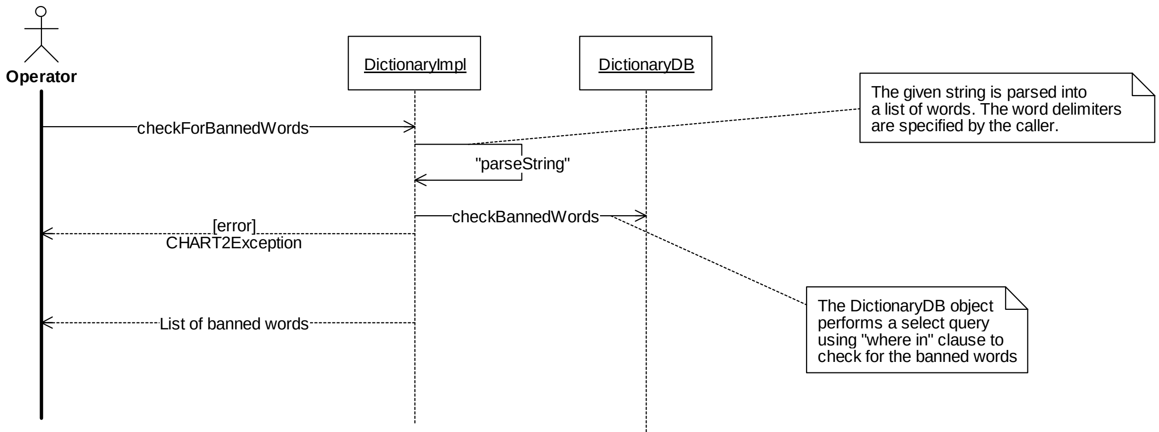

Figure 26. DictionaryImpl:checkForBannedWords (Sequence Diagram)

3.5.2.6 DictionaryImpl:getApprovedWords (Sequence Diagram)

The list of approved words in the dictionary is read from the database and returned to the operator. Access is denied to any operator without the “Manage Dictionary” privilege.

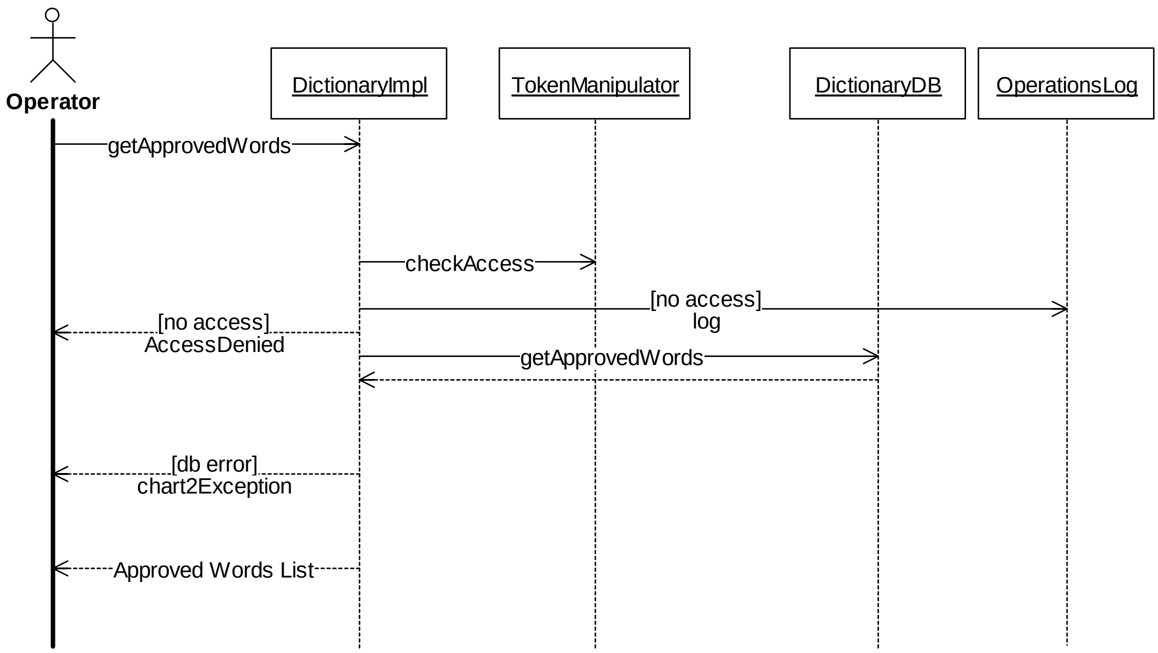

Figure 27. DictionaryImpl:getApprovedWords (Sequence Diagram)

3.5.2.7 DictionaryImpl:getBannedWords (Sequence Diagram)

The list of banned words in the dictionary is read from the database and returned to the operator. Access is denied to any operator without the “Manage Dictionary” privilege.

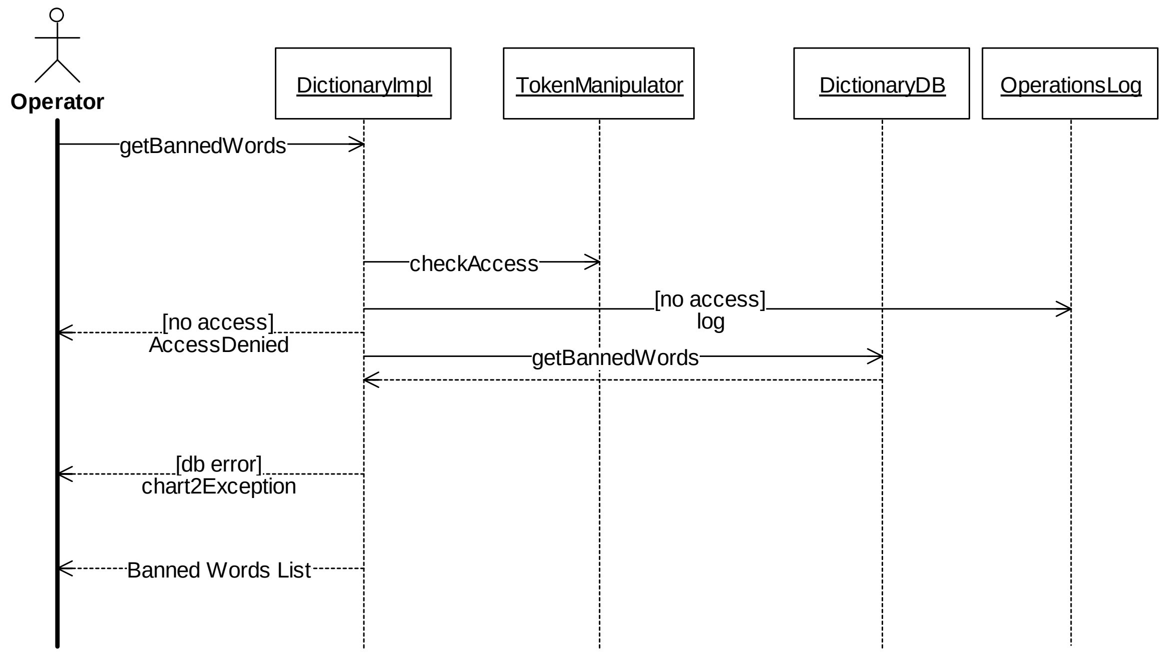

Figure 28. DictionaryImpl:getBannedWords (Sequence Diagram)

3.5.2.8 DictionaryImpl:PerformApprovedWordsCheck (Sequence Diagram)

The string provided by the operator is scanned for any words that are not present in the approved words dictionary database. Any character from the given set of delimiters is taken to be a valid delimiter of words in the string. For each word not present in the approved word list, a list of suggested words is formulated. The suggested words are those in the approved words dictionary, that have close lexical match with the disapproved word.

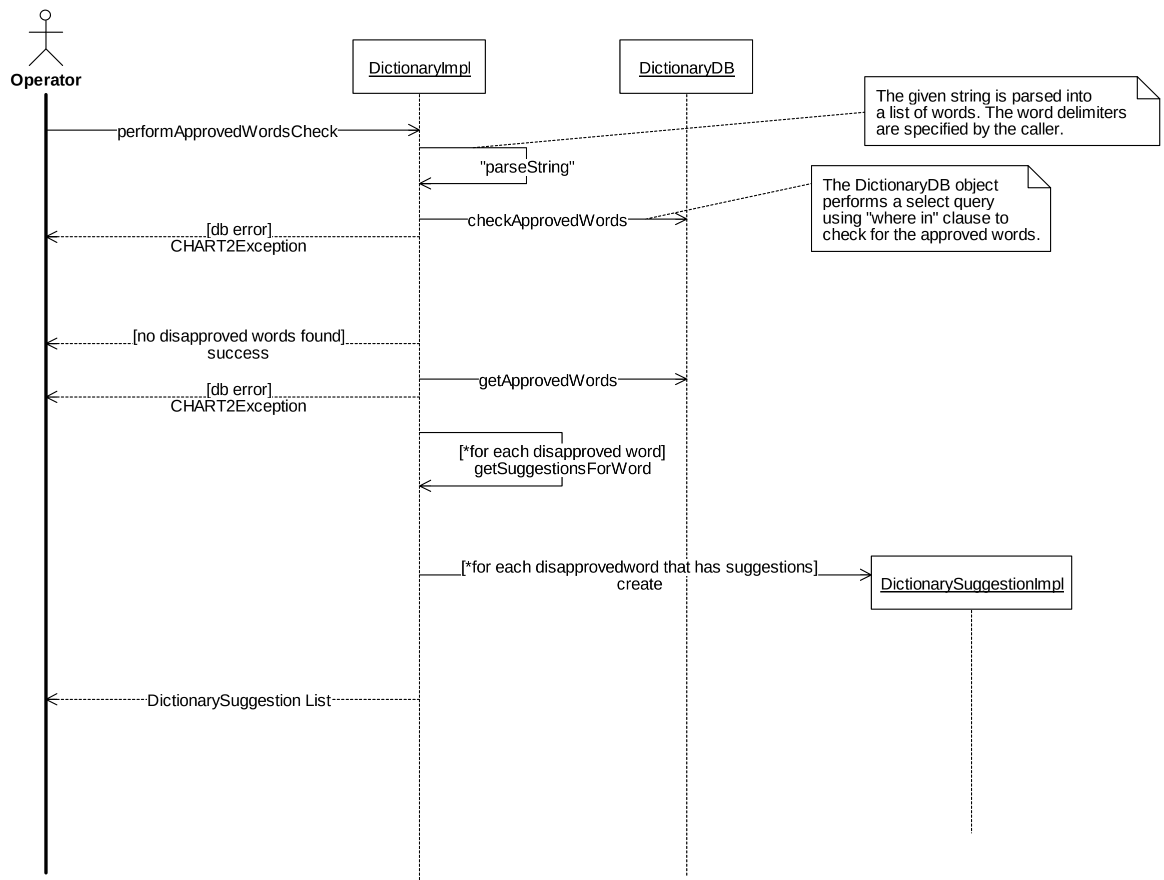

Figure 29. DictionaryImpl:PerformApprovedWordsCheck (Sequence Diagram)

3.5.2.9 DictionaryImpl:removeApprovedWordList (Sequence Diagram)

The given list of words is removed from the approved words dictionary database. The removed words are then communicated to the dictionary event consumers by invoking the push operation. Access is denied to any operator without the “Manage Dictionary” privilege.

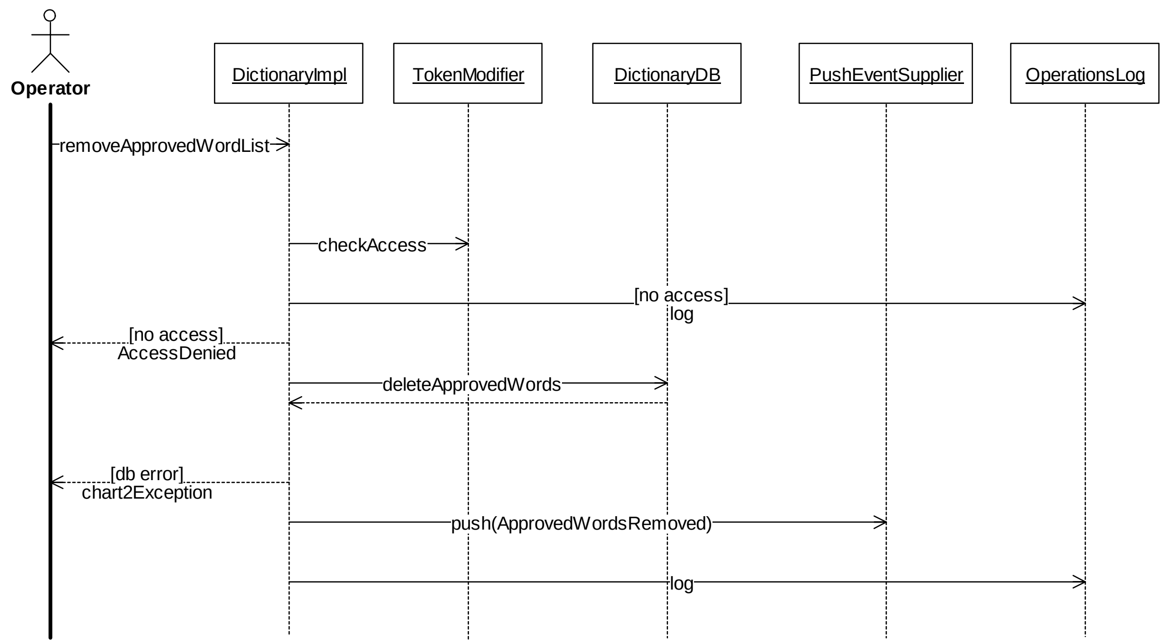

Figure 30. DictionaryImpl:removeApprovedWordList (Sequence Diagram)

3.5.2.10 DictionaryImpl:removeBannedWordList (Sequence Diagram)

The given list of words is removed from the banned words dictionary database. The removed words are then communicated to the dictionary event consumers by invoking the push operation. Access is denied to any operator without the “Manage Dictionary” privilege.

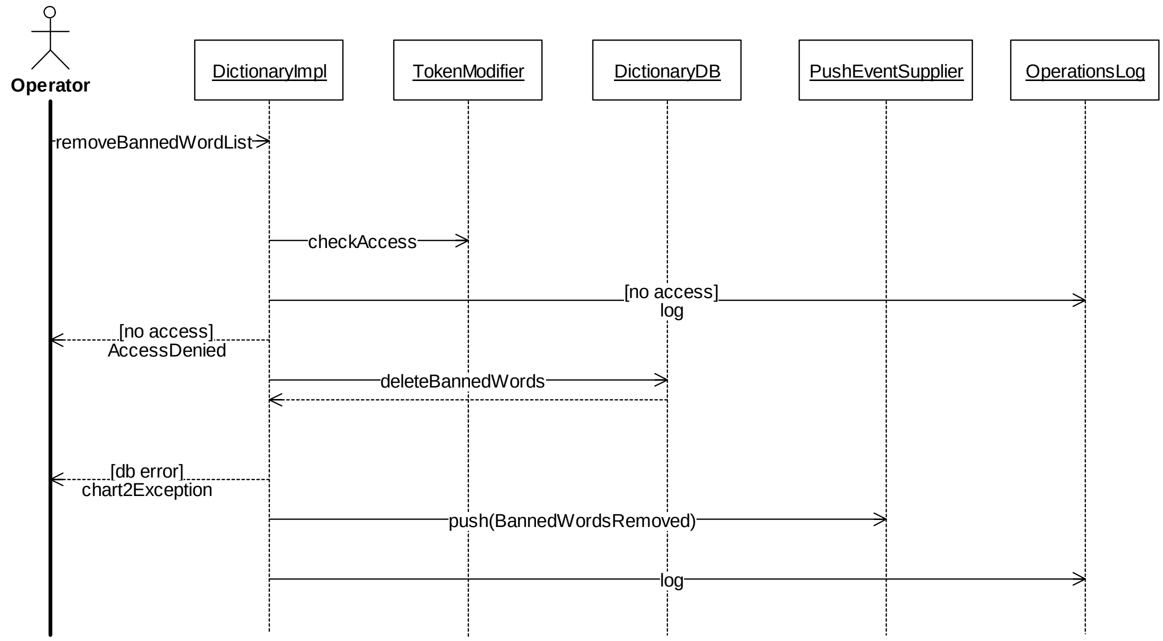

Figure 31. DictionaryImpl:removeBannedWordList (Sequence Diagram)

3.6 DMSControlModule

3.6.1 Classes

3.6.1.1 DMSControlClassDiagram (Class Diagram)

This Class Diagram shows the classes of the DMS Control Module. The DMS Control Module is an installable module that serves the DMS objects and DMSFactory to the rest of the CHART2 system. This diagram shows how the implementation of these CORBA interfaces rely on other supporting classes to perform their functions.

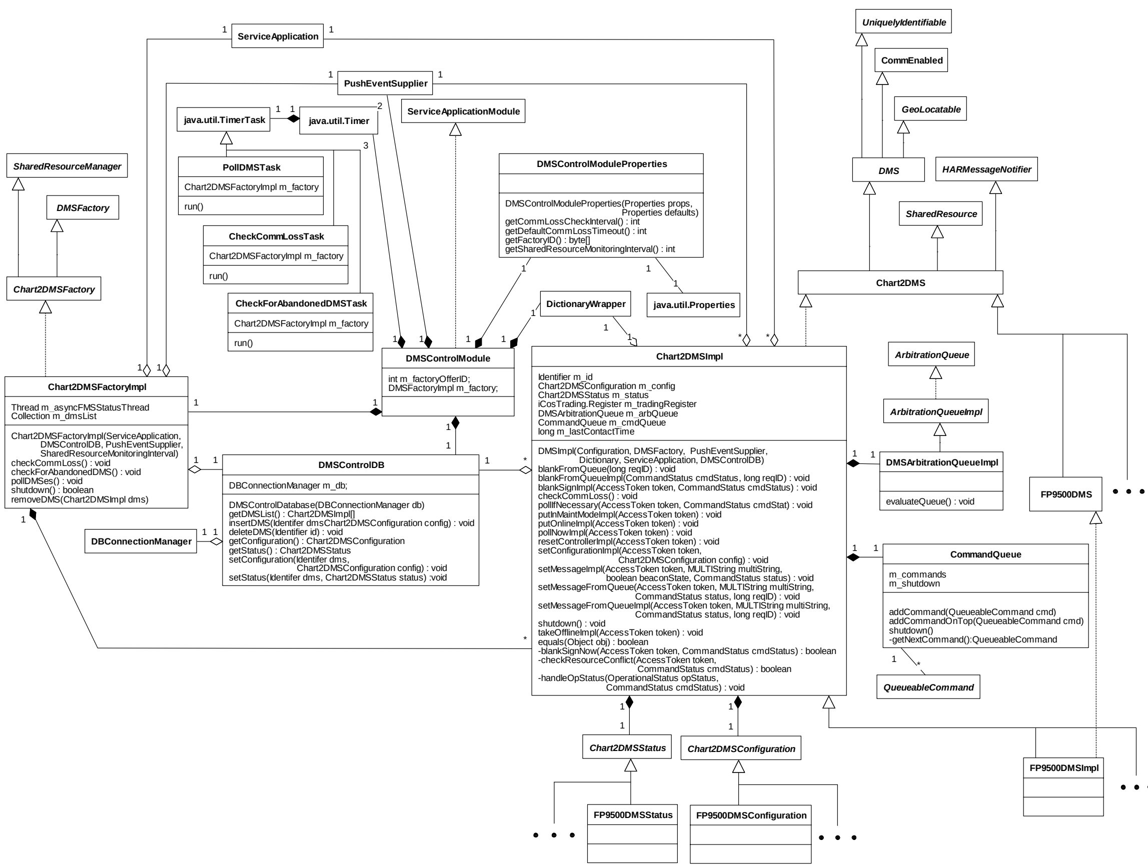

Figure 32. DMSControlClassDiagram (Class Diagram)

3.6.1.1.1 ArbitrationQueue (Class)

An ArbitrationQueue is a queue that arbitrates the usage of a device. The arbitration queue determines the proper message to be displayed on a device and switches the active message based on conditions present within the system.

For R1B2, the arbitration queue will have a single slot (and is therefore not really a queue). The device arbitration rules used by the arbitration queue in R1B2 rely on user rights and operations centers. When the arbitration queue's slot is in use, another message can only be added to the queue if the user adding the message is from the same operations center that is responsible for the existing message, or the user has a special functional right that allows this rule to be overridden.

The arbitration queue can be interrupted to keep it from performing its automated processing. This mode is used to allow maintenance on the device being arbitrated by the queue without having the queue's automatic processing interfere with the maintenance activities.

When an interrupted arbitration queue is taken out of its interrupted state through the use of the resume method, the arbitration queue evaluates the messages in the queue and restores the device to the proper state. For R1B2, the arbitration queue performs no special processing when resumed because the queue cleans itself when interrupted and does not allow new entries while interrupted.

3.6.1.1.2 ArbitrationQueueImpl (Class)

This class is an implementation of the ArbitrationQueue interface as defined by the IDL. This class arbitrates the usage of a messaging device (DMS or HAR) among multiple users. For R1B2, the arbitration algorithm is a "last in wins" scheme, where the last request to use the device being arbitrated overwrites any previous requests. When an arbitrated device is in use, the operations center of the requester is used to determine if the request will be allowed on the queue. Only a user from the same operations center that currently has a message on a device is allowed to overwrite a previous message. On exception to this is that users with a special functional right may override messages that were set from operations centers other than their own.

3.6.1.1.3 CHART2DMS (Class)

The CHART2DMS class extends the DMS interface and defines a more detailed interface to be used in manipulating the CHART II-specific DMS objects within CHART II. It provides a method for getting the DMSArbitrationQueue for a CHART II DMS, which can then be used by traffic events to provide input as to what each traffic event desires to be on the sign. It also provides a method to perform testing on a sign. This method can be extended by derived classes for specific models of signs, which know how to perform certain types of testing on their specific model of sign. CHART II business rules include concepts such as shared resrouces, arbitration queues, and linking devices usage to traffic events, concepts which go beyond what would be industry-standard DMS control.

3.6.1.1.4 CHART2DMSConfiguration (Class)

The CHART2DMSConfiguration class is an abstract class that extends the DMSConfiguration class to provide configuration information specific to CHART II processing. Such information includes how to contact the sign under CHART II software control, the default SHAZAM message for using the sign as a HAR Notifier, and the owning organization. Such data extends beyond what would be industry-standard configuration information for a DMS.

3.6.1.1.5 CHART2DMSFactory (Class)

The CHART2DMSFactory class extends the DMSFactory interface to provide additional CHART II specific capability. This factory creates CHART2DMS objects (extensions of DMS objects). It implements SharedResourceManager capbility control DMS objects as shared resources.

3.6.1.1.6 CHART2DMSFactoryImpl (Class)

The CHART2DMSFactoryImpl class provides an implementation of the CHART2DMSFactory interface (and DMSFactory interface) as specified in the IDL. The CHART2DMSFactoryImpl maintains a list of CHART2DMSImpl objects and is responsible for publishing DMS objects in the Trader on startup and as new DMS objects are created. Whenever a DMS is created or removed, that information is persisted to the database. This class is also responsible for performing the checks requested by the timer tasks: to poll the DMS devices and to look for DMS devices with timeout exceeded or with no one logged in at the controlling operations center.

3.6.1.1.7 CHART2DMSImpl (Class)

The CHART2DMSImpl class provides an implementation of the CHART2DMS interface, and by extension the DMS, SharedResource, HARMessageNotifier, CommEnabled, GeoLocatable, and UniquelyIdentifiable interfaces, as specified by the IDL. The CHART2DMSImpl contains a CommandQueue object that is used to sequentially execute long running operations (field communications to the device) in a thread separate from the CORBA request threads, thus allowing quick initial responses. The CHART2DMSImpl also contains a DMSArbitrationQueueImpl, which handles requests from TrafficEvents to display or remove messages from the signs in online mode. The DMSArbitrationQueueImpl validates and arbitrates these requests and makes calls into the CHART2DMSImpl, which then translates the requests into appropriate QueueableCommand objects (subclasses of QueueableCommand) ands adds them to the CommandQueue. The CHART2DMSImpl contains *Impl methods that map to each method specified in the IDL, including requests to put a message on the sign or remove a message (in maintenance mode only), put the sign online, offline, or in maintenance mode, or to change (set) the configuration of the sign. All of these requests require (or potentially require) field communications to the device, so each request is stored in a specific subclass of QueueableCommand and added to the CommandQueue. The queueable command objects simply call the appropriate CHART2DMSImpl method as the command is executed by the CommandQueue in its thread of execution. The CHART2DMSImpl also contains methods called by the CHART2DMSFactory to support the timer tasks of the DMS Service: to poll the DMS devices and to look for DMS devices with timeout exceeded or with no one logged in at the controlling operations center. This class contains a DMSConfiguration object and DMSStatus object, which are used store the configuration and status of the sign, and the it also contains a lastContactTime value, used for polling and for detecting communications timeouts.

3.6.1.1.8 CHART2DMSStatus (Class)

The CHART2DMSStatus class is an abstract class that extends the DMSStatus class to provide status information specific to CHART II processing, such as information on the controlling operations center for the sign. This data extends beyond what would be industry-standard status information for a DMS.

3.6.1.1.9 CheckCommLossTask (Class)

The CheckCommLossTask class is responsible for determining when communications to a DMS device have been down long enough to decide that the sign is or should be blank or considered to be blank. The anticipated time interval for making such a determination is on the order of ten minutes (however, this task is called much more frequently than that, so that the timeout can be detected soon after it has expired). This class implements the java.util.TimerTask interface, and as such it contains one method, run(), which is invoked by Java timer object on a regularly scheduled basis. This class contains a reference to the CHART2DMSFactoryImpl, which is called upon to actually check the DMS objects each time this task is called.

3.6.1.1.10 CheckForAbandonedDMSTask (Class)

The CheckForAbandonedDMSTask class is responsible for detecting any DMS device with a message on it that has no one logged in at the controlling operations center. This would only occur as a result of an anomaly—such as a reboot of a user's machine—because during a normal CHART II logout attempt, the logout is prohibited by CHART II system if the the user is the last user on his/her operations center *and* that operations center is controlling a sign. However, since anomalies happen, this task runs periodically to look for abandoned DMS devices. This class implements the java.util.TimerTask interface, and as such it contains one method, run(), which is invoked by Java timer object on a regularly scheduled basis. This class contains a reference to the CHART2DMSFactoryImpl, which is called upon to actually check the DMS objects and controlling operations centers of each DMS every time this task is called.

3.6.1.1.11 CommandQueue (Class)

The CommandQueue class provides a queue for QueuableCommand objects. The CommandQueue has a thread that it uses to process each QueuableCommand in a first in first out order. As each command object is pulled off the queue by the CommandQueue's thread, the command object's execute method is called, at which time the command performs its intended task.

3.6.1.1.12 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.6.1.1.13 DMSArbitrationQueueImpl (Class)

The DMSArbitrationQueueImpl class is a derivation of the ArbitrationQueue class that is customized to support DMS objects. It basically operates as a generic ArbitrationQueueImpl, but it contains a DMS-specific implementation of the ArbitrationQueue's abstract evaluateQueue method. For this release, the only distinct features of this method (as compared with the HARArbitrationQueueImpl's version) is that the setMessageFromQueue method it calls must be on a DMS type of object, and the parameters used in calling it for a DMS class is different from calling the setMessageFromQueue method of the HAR class.

3.6.1.1.14 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two separate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.6.1.1.15 DictionaryWrapper (Class)

This singleton class provides a wrapper for the system dictionary that provides automatic location of the dictionary and automatic re-discovery should the dictionary reference return an error. This class also allows for built-in fault tolerence by automatically failing over to a "working" dictionary without the user of this class being aware that this being done. In addition, this class defers the discovery of the Dictionary until its first use, thus eliminating a start-up dependency for modules that use the dictionary.

This class delegates all of its method calls to the system dictionary using its currently known good reference to the system dictionary. If the current reference returns a CORBA failure in the delegated call, this class automatically switches to another reference. When there are no good references (as is true the first time the object is used), this class issues a trader query to (re)discover the published Dictionary objects in the system. During a method call, the trader will be queried at most one time and under normal circumstances (other than the first use) the trader will not be queried at all.

3.6.1.1.16 DMSControlDB (Class)

The DMSControlDB class provides an interface between the DMS service and the database used to persist the DMS objects and their configuration and status in the database. It contains a collection of methods that perform database operations on tables pertinent to DMS Control. The class is constructed with a DBConnectionManager object, which manages database connections. Methods exist to insert and delete DMS objects from the database, and to get and set their configuration and status information. All information about a sign is persisted, including its current displayed message, communications status, and time of last contact, so that a momentary glitch or restart of the software will not interrupt messages on signs.

3.6.1.1.17 DMSControlModuleProperties (Class)

The DMSControlModuleProperties class is used to provide access to properties used by the DMS Control Module. This class wraps properties that are passed to it upon construction. It adds its own defaults and provides methods to extract properties specific to the DMS Control Module.

3.6.1.1.18 DMS (Class)

The DMS class defines an interface to be used in manipulating Dynamic Message Sign (DMS) objects within CHART II. It specifies methods for setting messages and clearing messages from a sign (in maintenance mode), polling a sign, changing the configuration of a sign, and reseting a sign. (Setting messages on a sign in online mode are not accomplished by manipulating a DMS directly; that is accomplished by manipulating traffic events, which interfaces with the DMSArbitrationQueue of a sign. This activity involves the DMS extension, CHART2DMS, which defines interactions with signs under CHART II business rules.)

3.6.1.1.19 DMSControlModule (Class)

The DMSControlModule class is is the service module for the DMS devices and a DMS factory. It implements the ServiceApplicationModule interface. It creates and serves a single DMSFactoryImpl object, which in turn serves zero or more CHART2DMSImpl objects.

3.6.1.1.20 DMSFactory (Class)

The DMSFactory class specifies the interface to be used to create DMS objects within the CHART II system. It also provides a method to get a list of DMS devices currently in the system.

3.6.1.1.21 FP9500DMS (Class)

The FP9500DMS class extends the CHART2DMS interface and defines a more detailed interface to be used in manipulating FP9500 models of DMS signs. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific DMS control. For instance, the FP9500DMS has a performPixelTest method, which knows how to invoke and interpret a pixel test as supported by the FP9500 model DMS.

3.6.1.1.22 FP9500DMSConfiguration (Class)

The FP9500Configuration class is an abstract class that extends the CHART2DMSConfiguration class to provide configuration information specific to an FP9500 model of DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information.

3.6.1.1.23 FP9500DMSImpl (Class)

The FP9500DMSImpl class provides a specific implementation to implement the FP9500DMS interface, providing any specific functionality unique to this brand and model of sign. This class is exemplary of a whole suite of implementation classes that may be created, on a case-by-case basis, to support specific capabilities of speciifc brands and models of signs.

3.6.1.1.24 FP9500DMSStatus (Class)

The FP9500DMSStatus class provides additional storage for status information unique to the FP9500 model of sign. It is exemplary of potentially a whole suite of CHART2DMSStatus subclasses specific to a specific brand and model of sign.

3.6.1.1.25 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.6.1.1.26 HARMessageNotifier (Class)

The HARMessageNotifier class specifies an interface to be implemented by devices that can be used to notify the traveler to tune in to a radio station to hear a traffic message being broadcast by a HAR. A HARMessageNotifier is directional and allows users of the device to better determine if activation of the device is warranted for the message being broadcast by the HAR. This interface can be implemented by SHAZAMs and by DMS devices that are allowed to provide a SHAZAM-like message in the absence of any more useful messages to display.

3.6.1.1.27 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its “defaults”; this second property list is searched if the property key is not found in the original property list.

3.6.1.1.28 java.util.Timer (Class)

This class provides asynchronous execution of tasks that are scheduled for one-time or recurring execution.

3.6.1.1.29 java.util.TimerTask (Class)

This class is an abstract base class which can be scheduled with a timer to be executed one or more times.

3.6.1.1.30 PollDMSTask (Class)

The PollDMSTask class is responsible for polling all the DMS devices. This class implements the java.util.TimerTask interface, and as such it contains one method, run(), which is invoked by Java timer object on a regularly scheduled basis. This class contains a reference to the CHART2DMSFactoryImpl, which is called upon to request each DMS to poll itself (its poll interval has expired) each time this task is called.

3.6.1.1.31 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier’s push rate.

3.6.1.1.32 QueueableCommand (Class)

A QueueableCommand is an interface used to represent a command that can be placed on a CommandQueue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed. This interface must be implemented by any device command in order that it may be queued on a CommandQueue. The CommandQueue driver calls the execute method to execute a command in the queue and a call to the interuppted method is made when a CommandQueue is shut down.

3.6.1.1.33 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.6.1.1.34 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.6.1.1.35 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.6.1.1.36 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn’t have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.6.1.1.37 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.6.1.2 QueueableCommandClassDiagram (Class Diagram)

This class diagram shows the classes derived from QueueableCommand necessary for DMS Control. A class exists for each type of command that can be executed asynchronously on a DMS object.

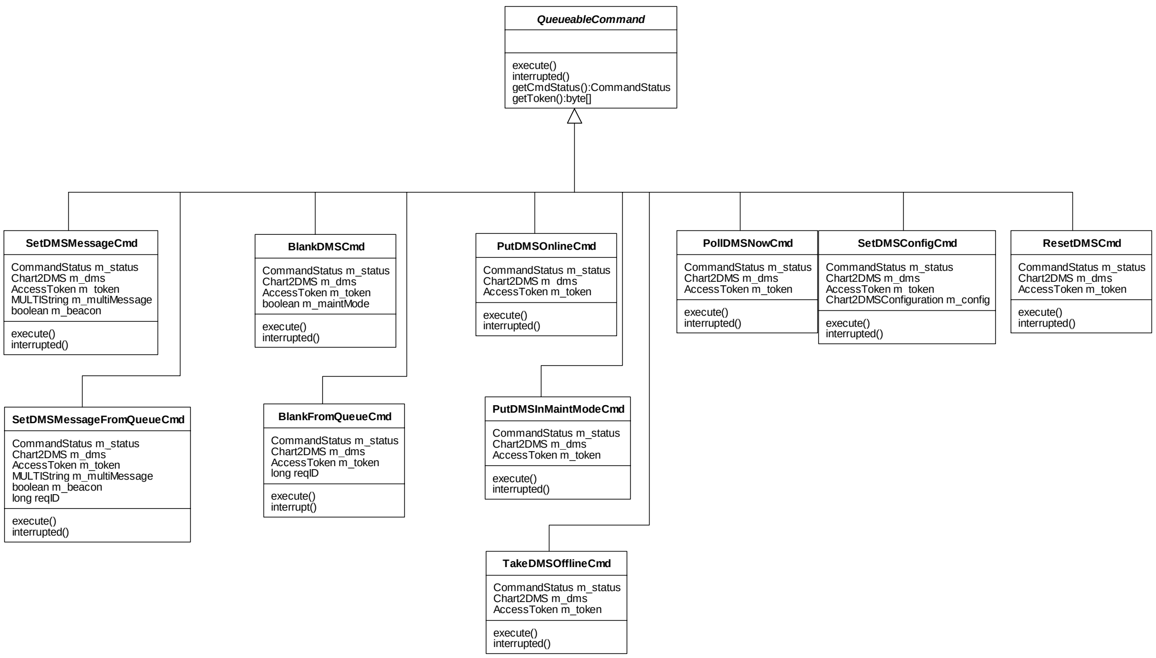

Figure 33. QueueableCommandClassDiagram (Class Diagram)

3.6.1.2.1 BlankDMSCmd (Class)

The BlankDMSCmd class is a QueueableCommand subclass that contains data necessary to send a request to a CHART2DMSImpl to blank the sign in maintenance mode. It is created by the CHART2DMSImpl during successful processing of its blankSign method. When the CommandQueue invokes the execute method of this class, it merely calls the blankSignImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.1.2.2 BlankFromQueueCmd (Class)

The BlankDMSFromQueueCmd class is a QueueableCommand subclass that contains data necessary to send a request to a CHART2DMSImpl to blank the sign during normal operations (online mode). It is created by the CHART2DMSImpl during successful processing of its blankFromQueue method. When the CommandQueue invokes the execute method of this class, it merely calls the blankFromQueueImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.1.2.3 PollDMSNowCmd (Class)

The PollDMSNowCmd class is a QueueableCommand subclass that contains data necessary to send a request to a CHART2DMSImpl to poll its device. It is created by the CHART2DMSImpl during successful processing of its pollNow method in maintenance mode (triggered by a user request) or during processing of the pollIfNecessary method (triggered by the automatic polling of the PollDMSTask object). When the CommandQueue invokes the execute method of this class, it merely calls the pollNowImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.1.2.4 PutDMSInMaintModeCmd (Class)

The PutDMSInMaintModeCmd class is a QueueableCommand subclass that contains data necessary to send a request to a CHART2DMSImpl to put the sign in maintenance mode (from either offline or online mode). It is created by the CHART2DMSImpl during successful processing of its putDMSInMaintMode method. When the CommandQueue invokes the execute method of this class, it merely calls the putDMSInMaintModeImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.1.2.5 PutDMSOnlineCmd (Class)

The PutDMSOnlineCmd class is a QueueableCommand subclass which contains data necessary to send a request to a CHART2DMSImpl to put the sign online (from either offline or maintenance mode). It is created by the CHART2DMSImpl during successful processing of its putDMSOnline method. When the CommandQueue invokes the execute method of this class, it merely calls the putDMSOnlineImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.1.2.6 QueueableCommand (Class)

A QueueableCommand is an interface used to represent a command that can be placed on a CommandQueue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed. This interface must be implemented by any device command in order that it may be queued on a CommandQueue. The CommandQueue driver calls the execute method to execute a command in the queue and a call to the interuppted method is made when a CommandQueue is shut down.

3.6.1.2.7 ResetDMSCmd (Class)

The ResetDMSCmd class is a QueueableCommand subclass that contains data necessary to send a request to a CHART2DMSImpl to put reset the sign (in maintenance mode only). It is created by the CHART2DMSImpl during successful processing of its resetController method. When the CommandQueue invokes the execute method of this class, it merely calls the resetControllerImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.1.2.8 SetDMSConfigCmd (Class)

The SetDMSConfigCmd class is a QueueableCommand subclass that contains data necessary to send a request to a CHART2DMSImpl to update its configuration (in maintenance mode only). It is created by the CHART2DMSImpl during successful processing of its setConfiguration method. When the CommandQueue invokes the execute method of this class, it merely calls the setConfigurationImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.1.2.9 SetDMSMessageCmd (Class)

The SetDMSMessageCmd class is a QueueableCommand subclass that contains data necessary to send a request to a CHART2DMSImpl to put a message on the sign in maintenance mode. It is created by the CHART2DMSImpl during successful processing of its setMessage method. When the CommandQueue invokes the execute method of this class, it merely calls the setDMSMessageImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.1.2.10 SetDMSMessageFromQueueCmd (Class)

The SetDMSMessageFromQueueCmd class is a QueueableCommand subclass that contains data necessary to send a request to a CHART2DMSImpl to put a message on the sign during normal operations (online mode). It is created by the CHART2DMSImpl during successful processing of its setMessage method. When the CommandQueue invokes the execute method of this class, it merely calls the setDMSMessageFromQueueImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.1.2.11 TakeDMSOfflineCmd (Class)

The TakeDMSOfflineCmd class is a QueueableCommand subclass which contains data necessary to send a request to a CHART2DMSImpl to put the sign offline (from either online or maintenance mode). It is created by the CHART2DMSImpl during successful processing of its takeDMSOffline method. When the CommandQueue invokes the execute method of this class, it merely calls the takeDMSOfflineImpl method of the appropriate CHART2DMSImpl object with the data stored within this class.

3.6.2 Sequence Diagrams

3.6.2.1 DMSControlModule:ActivateHARNotice (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object responds to a request to be used as a SHAZAM by a HAR. This method is called by the HAR's activateMessageNotifier method. The operator (making the original HAR request) must have proper functional rights for the sign, and the sign must be online. This method creates a HARNotifierArbQueueEntry and adds it to its own ArbitrationQueue via the ArbitrationQueue's addEntry command. (The ArbitrationQueue will not replace a "real" DMS message with a SHAZAM message. However, for this build if the request is rejected by the ArbitrationQueue the request is completely tossed out – it is not queued up to wait for an opportunity later when the other message goes away.) If the ArbitrationQueue accepts the request, it may eventually by processed by the DMS's setMessageFromQueue method.

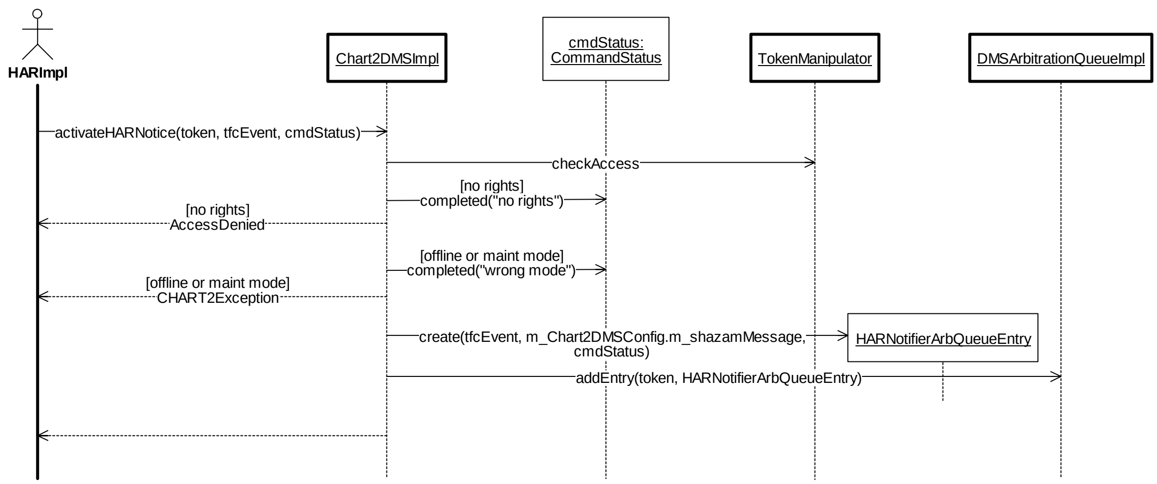

Figure 34. DMSControlModule:ActivateHARNotice (Sequence Diagram)

3.6.2.1.1 DMSControlModule:BlankFromQueue (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object processes a request to blank its message while it is online. (For blanking messages in maintenance mode, see blankSign.) This sequence is actually initiated in the ArbitrationQueue, when it determines that the current message no longer belongs on the sign, and it has no other message to replace it. The ArbitrationQueue's evaluateQueue method calls this method. The DMS must still be online. There is no operator associated with this request, no functional rights to verify, and no operator-monitored CommandStatus object to update. A BlankFromQueueCmd (a QueueableCommand) is created and added to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. When the CommandQueue is ready, it executes the BlankFromQueueCmd, which calls the BlankFromQueueImpl method, also shown on this diagram. The blankFromQueueImpl method simply calls blankSignNow, and reports success or failure to the ArbitrationQueue via the requestFailed or requestSucceeded method (at which time the ArbitrationQueue may re-evaluate its own queue and request another change to the sign). Althought there is no CommandStatus object directly communicating status of this operation, the ArbitrationQueue still updates the TrafficEvent(s) that did have control of the sign when the sign is successfully blanked.

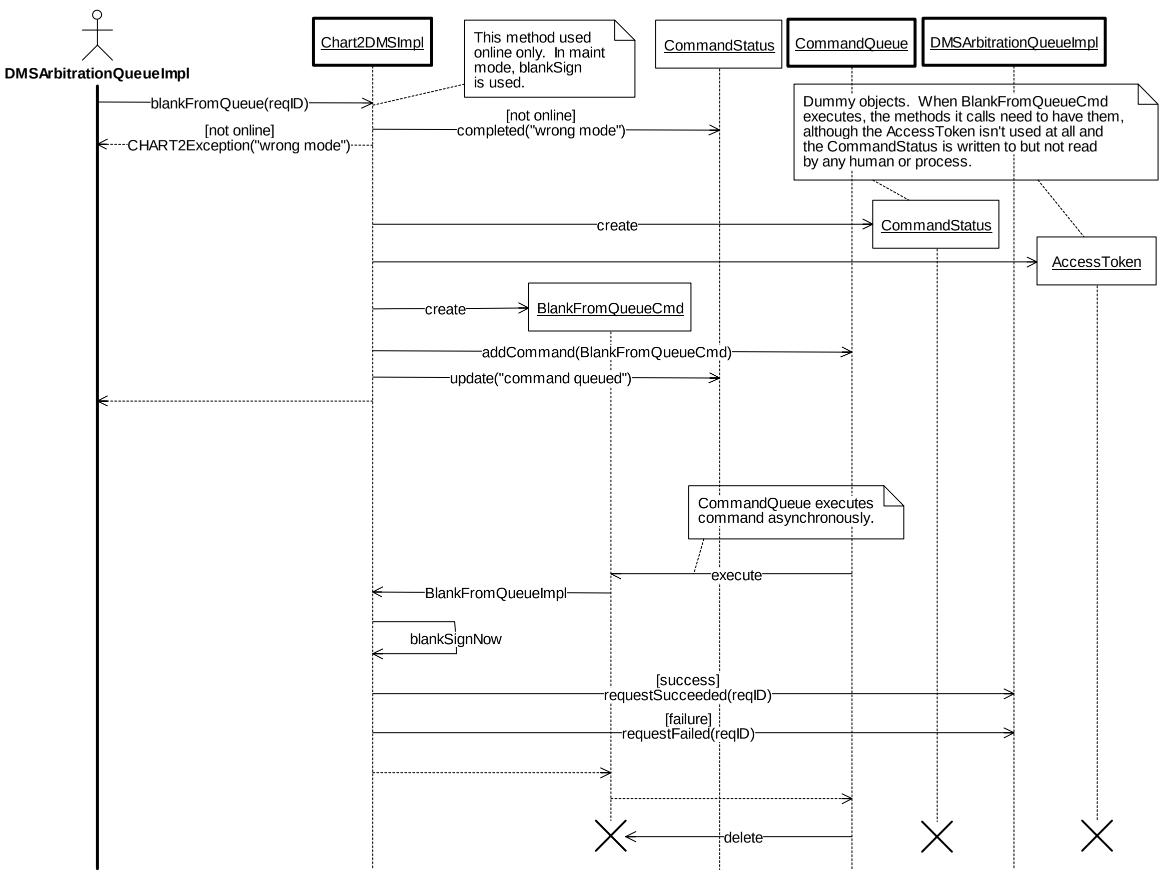

Figure 35. DMSControlModule:BlankFromQueue (Sequence Diagram)

3.6.2.1.2 DMSControlModule:BlankSign (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object processes a request to blank its message in maintenance mode. (The analogous method in online is blankFromQueue.) The DMS must be in maintenance mode, the requesting operator must have proper functional rights, and if there is a message on the sign from another operations center, the user must have override authority. This method creates a BlankDMSCmd (a QueueableCommand) and adds it to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. Requests to communicate with the sign are processed on a first-come, first-served basis. When the CommandQueue is ready, it executes the BlankDMSCmd, which calls the blankSignImpl method. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

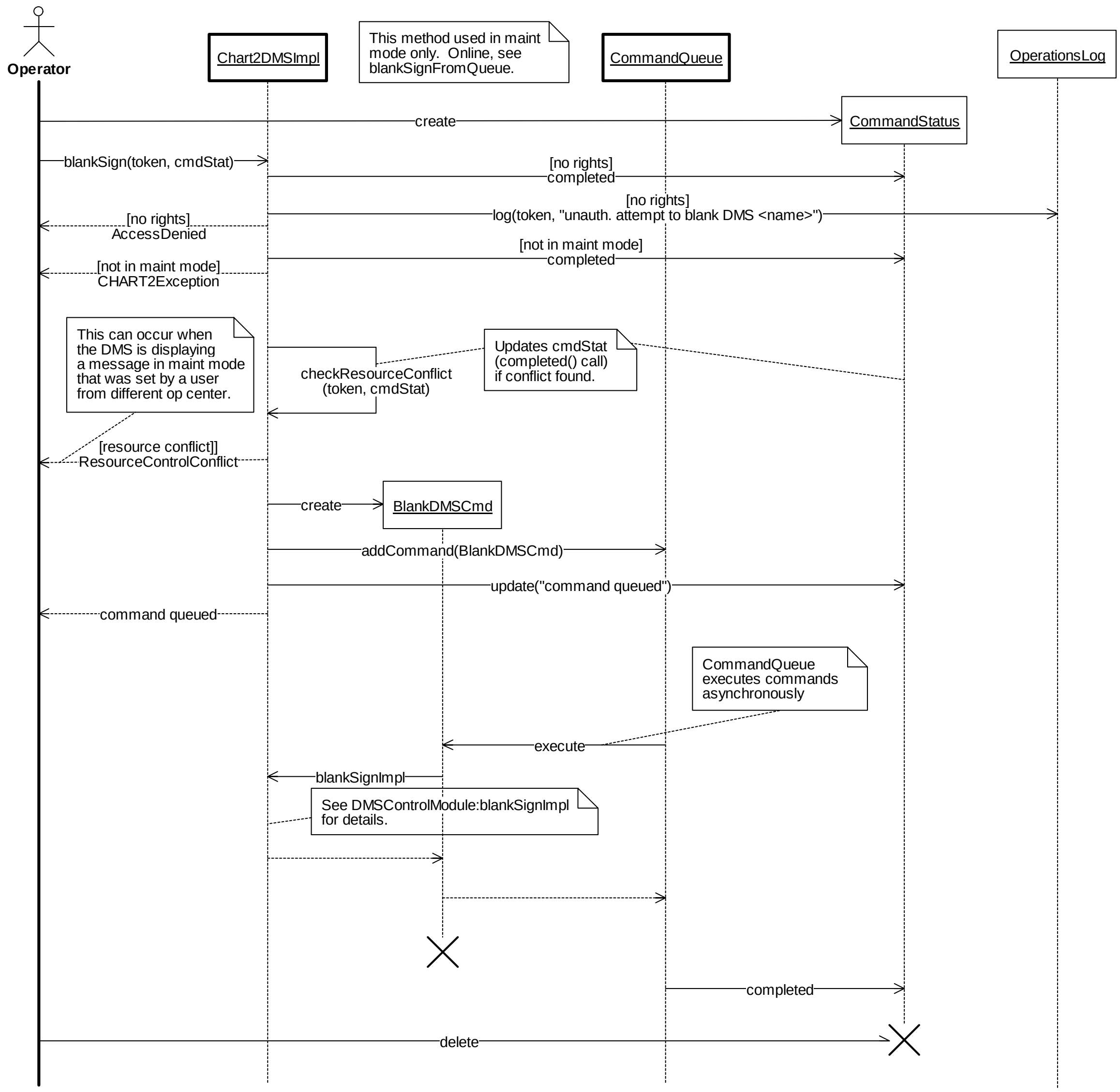

Figure 36. DMSControlModule:BlankSign (Sequence Diagram)

3.6.2.2 DMSControlModule:BlankSignImpl (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object executes a command to blank its message in maintenance mode. (The analogous method in online mode is blankFromQueueImpl.) An operator request to blank the sign has already been received and pre-processed by the blankSign method. When the blankSignImpl method runs, it checks that the DMS is still in maintenance mode (a previously queued command could have changed it), that the user has rights, and that there is no resource conflict (a previously queued command could have written a message from an operator at another operations center). Assuming no problems, the method blankSignNow is called to request FMS to actually change the sign, update the database, and handle any status change, and push a CurrentDMSStatus event into the event channel, so that any user (with rights) can immediately see that the sign is now blank. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

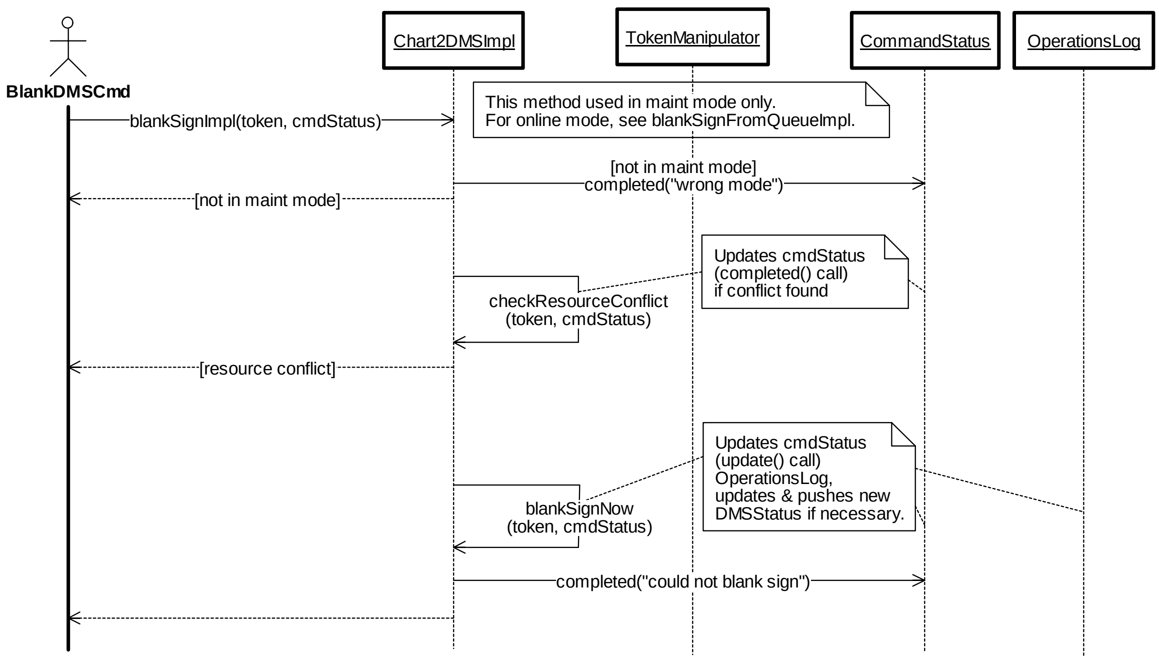

Figure 37. DMSControlModule:BlankSignImpl (Sequence Diagram)

3.6.2.3 DMSControlModule:BlankSignNow (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object actually blanks the sign. This is a utility method called at many points during DMS operations. The sign must be blanked when requested by the user in maintenance mode, when implicitly requested when online by removing a message, when changing modes (online, offline, maintenance mode), and when resetting the sign. This method blanks the sign by creating an empty message and requesting, via FMS, that the sign display the blank message. The method handleOpStatus handles and responds to any changes to the operational status of the sign (OK, comms failure, or hardware failure) reported by FMS during this operation. This method writes progress and status information to a CommandStatus object, so that progress can be monitored by the user (if any is associated with this operation – there is no user with an implicit request by the ArbitrationQueue to blank a sign while online). A CurrentDMSStatus event is pushed into the event channel, so that any user (with rights) can immediately see that the sign is now blank.

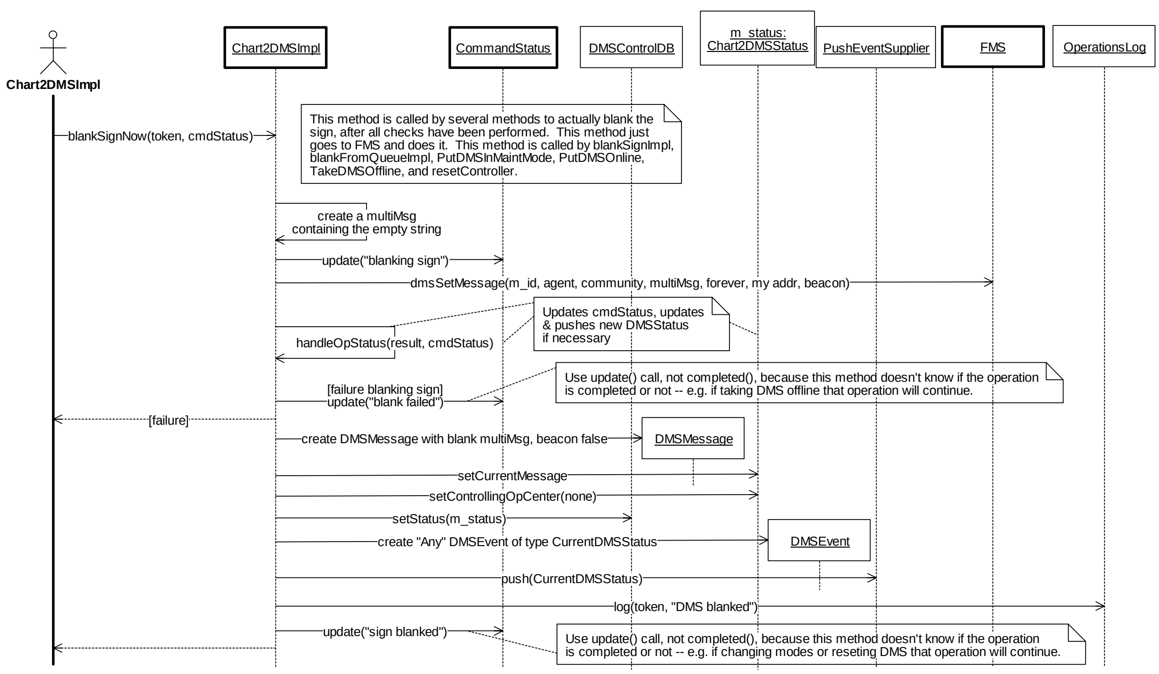

Figure 38. DMSControlModule:BlankSignNow (Sequence Diagram)

3.6.2.4 DMSControlModule:CheckResourceConflict (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object checks a sign for a resource conflict prior to performing some other sort of operation on it. This utility method is called from several other methods within the DMS service. If the DMS is currently displaying a message, and therefore has a controlling operations center, and it is not equal to the caller's operations center, and the user does not have override authority, there is a resource control conflict. Otherwise, there is not. If there is a resource control conflict, a message to this effect is written to the CommandStatus object, which may be monitored by the requesting user.

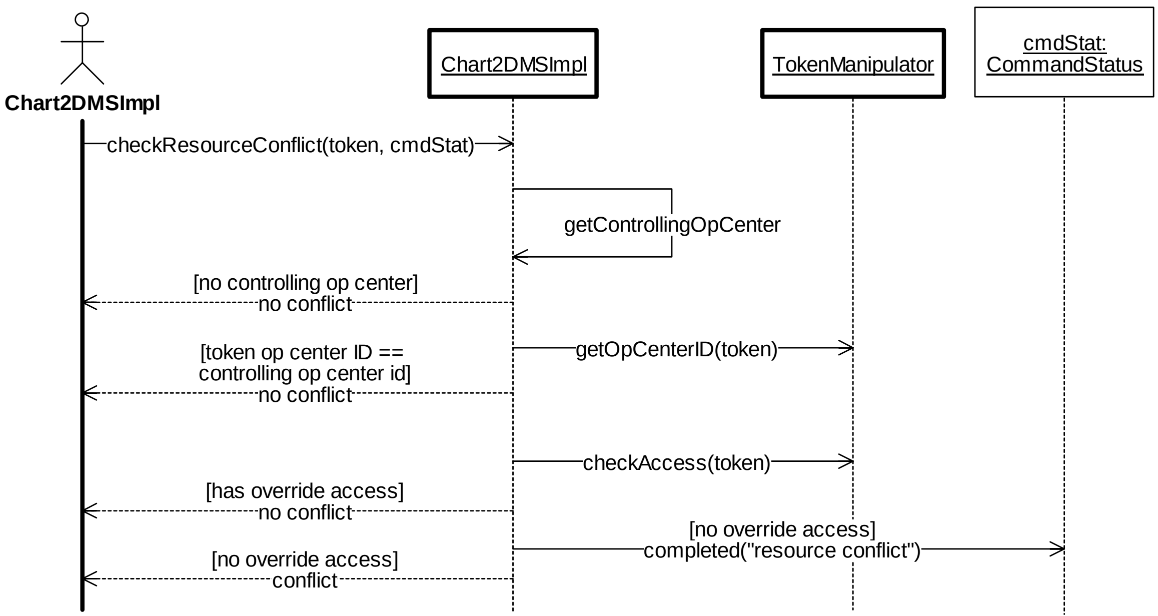

Figure 39. DMSControlModule:CheckResourceConflict (Sequence Diagram)

3.6.2.5 DMSControlModule:CreateDMS (Sequence Diagram)

This Sequence Diagram shows how the DMSFactoryImpl creates a new DMS on behalf of an operator. The operator must posess the proper functional rights to create a DMS. The request to create a new DMS contains all data necessary to create it in a DMSConfiguration object—most likely one of some specific subclass, such as FP9500DMSConfiguration (unless it is to be a truly generic CHART2DMS, one which has no extended capabilities, or one of a new type whose extended capabilities are not yet encoded in CHART II software). When a request to create DMS is received by the DMSFactory, the DMSControlDB is asked to create and persist it to the database. A (subclassed) CHART2DMSImpl object and its corresponding DMSArbitrationQueueImpl and CommandQueueImpl are created, and the CommandQueue thread is started. Information about the new DMS is also communicated to the FMS subsystem. The object is connected to the ORB and is ready for operations. A DMSAddedEvent is then pushed into the event channel. A DMS is initially in offline mode when it is created.

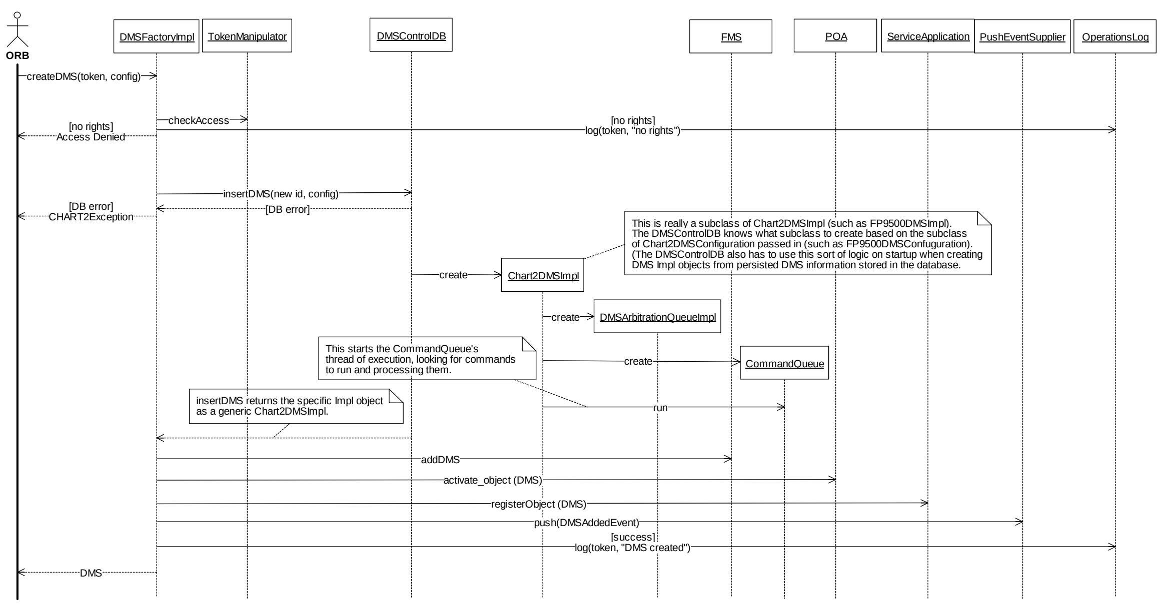

Figure 40. DMSControlModule:CreateDMS (Sequence Diagram)

3.6.2.6 DMSControlModule:DeactivateHARNotice (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object responds to a request to discontinue operation as a SHAZAM for a HAR. This method is called by the HAR's deactivateMessageNotifier method. The operator (ending the HAR message) must have proper functional rights for the sign, and the sign must be online. This method calls the ArbitrationQueue's removeEntry command to handle the request, and the ArbitrationQueue will respond (now or later) with another setMessageFromQueue or blankFromQueue request, as appropriate.

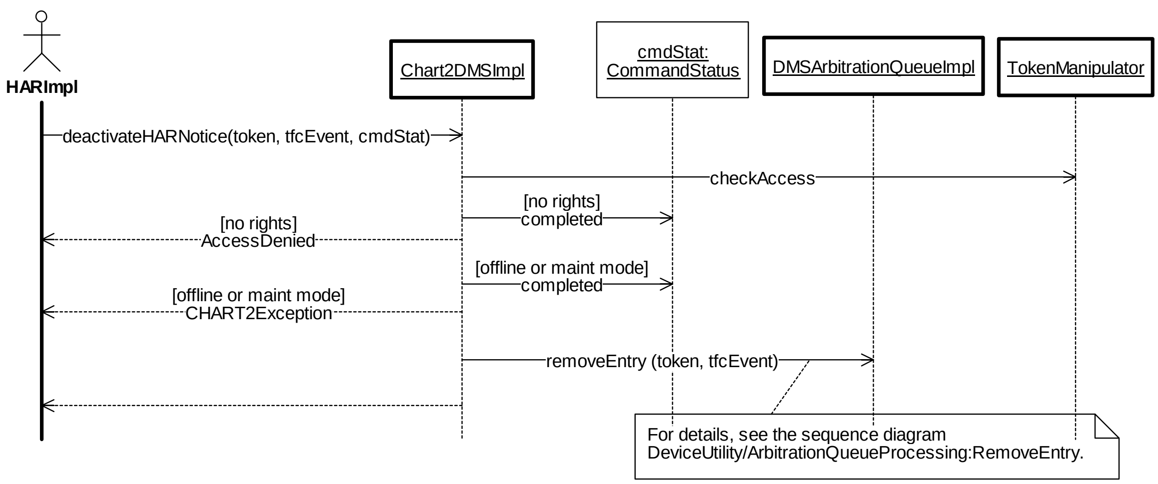

Figure 41. DMSControlModule:DeactivateHARNotice (Sequence Diagram)

3.6.2.7 DMSControlModule:GetConfiguration (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object responds to a request for its configuration. Its configuration is always maintained in current form in a CHART2DMSConfiguration object, so this object is just returned immediately.

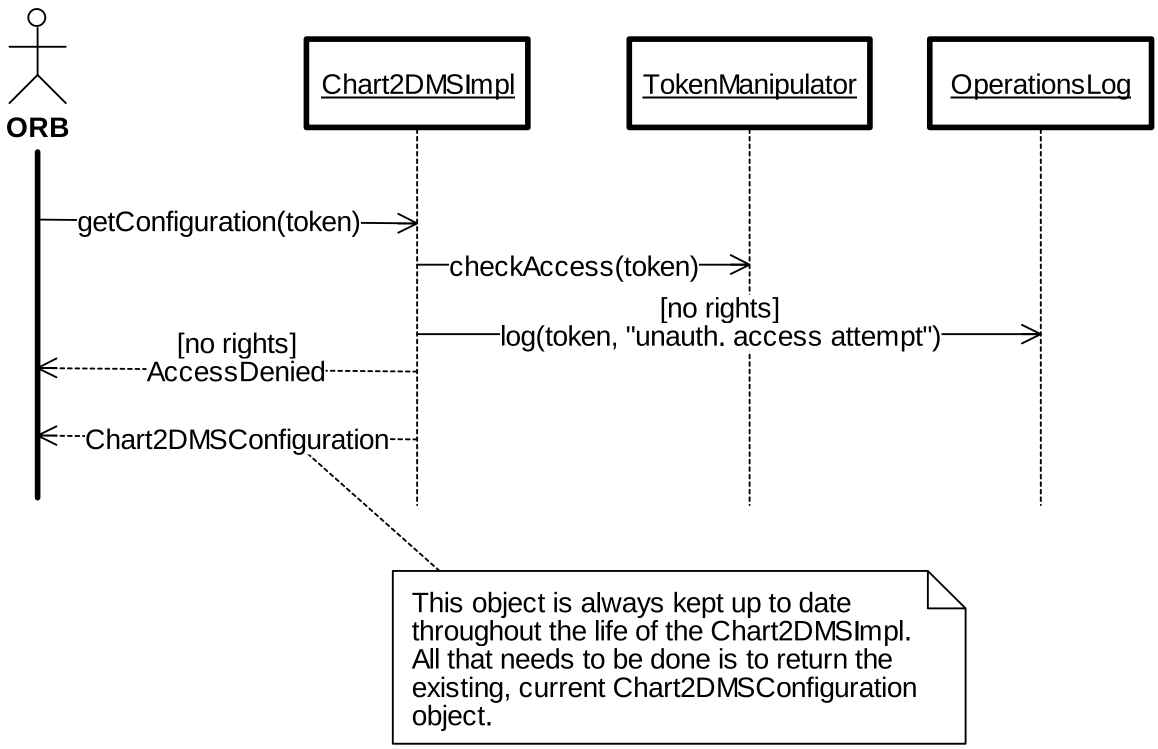

Figure 42. DMSControlModule:GetConfiguration (Sequence Diagram)

3.6.2.8 DMSControlModule:GetControlledResources (Sequence Diagram)

This Sequence Diagram shows how the CHART2DMSFactoryImpl handles a request to get a list of controlled resources for an operations center. The CHART2DMSFactoryImpl simply asks each CHART2DMSImpl for its controlling operations center, and if it matches the OperationsCenter in question, the DMS is added to a list. This list is returned to the caller.

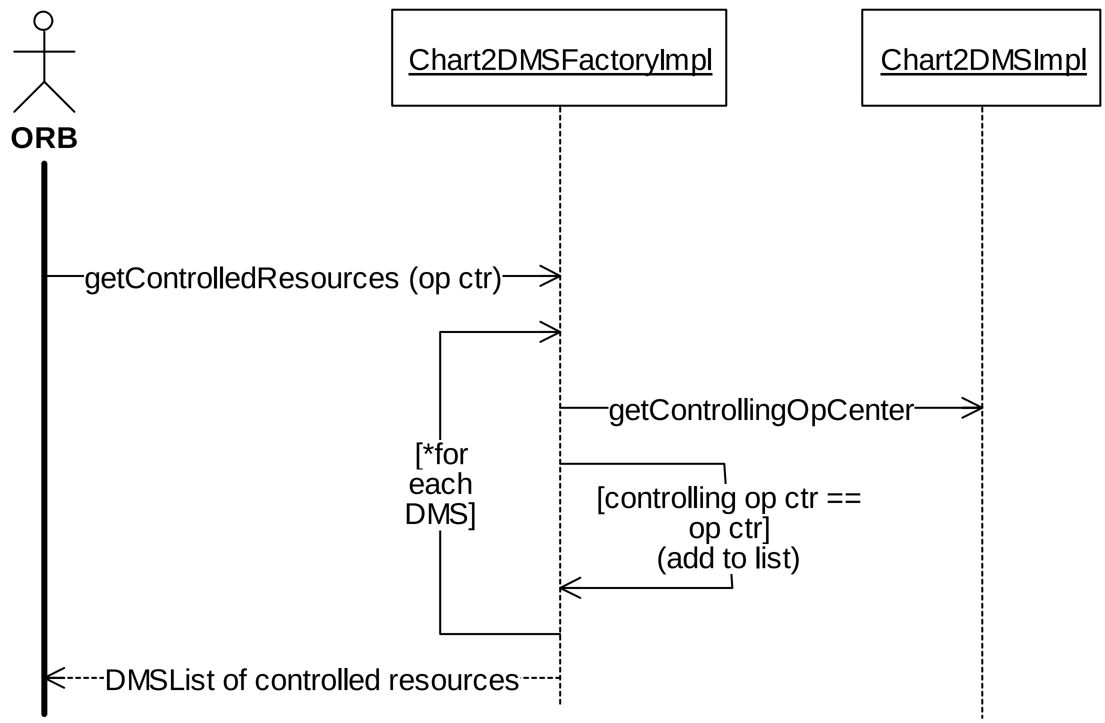

Figure 43. DMSControlModule:GetControlledResources (Sequence Diagram)

3.6.2.9 DMSControlModule:GetStatus (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object responds to a request for its status. Its status is always maintained in current form in a CHART2DMSStatus object, so this object is just returned immediately.

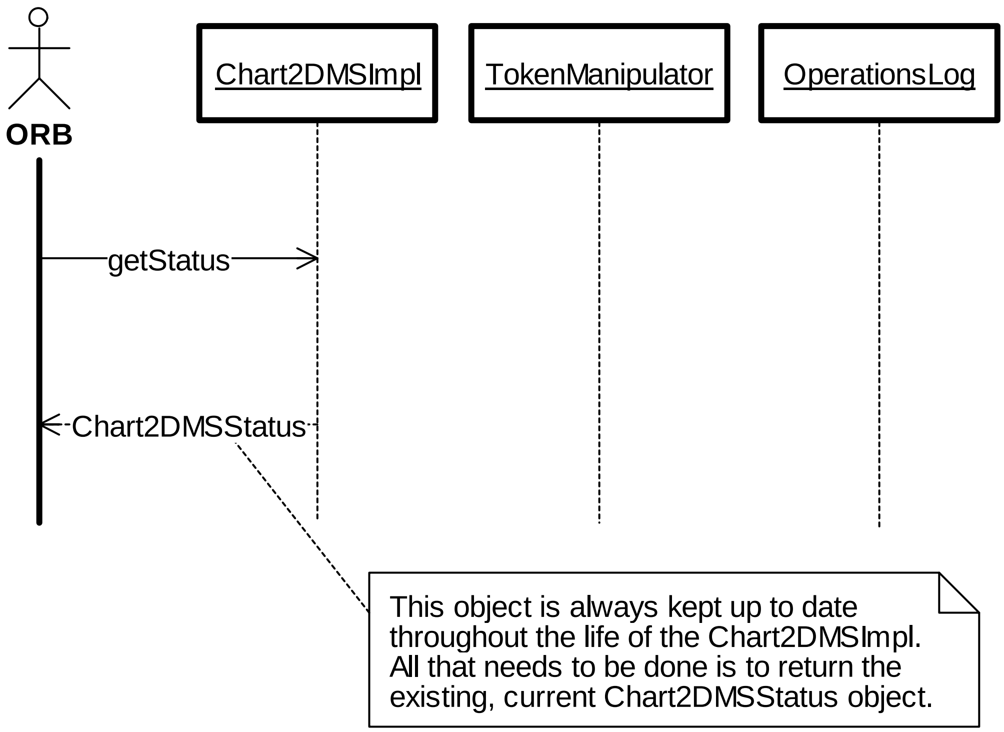

Figure 44. DMSControlModule:GetStatus (Sequence Diagram)

3.6.2.10 DMSControlModule:HandleOpStatus (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl handles the important task of detecting and responding to changes in its operational status (whether it is in “OK”, “COMM_FAILURE” or “HARDWARE_FAILURE” status). A DMS is normally “OK”, but falls into “COMM_FAILURE” when FMS reports that it cannot communicate with the device, and into “HARDWARE_FAILURE” when the FMS can communicate with the device but the device or FMS is detecting some sort of hardware problem with the device itself. At this point, HARDWARE_FAILURE and COMM_FAILURE are treated virtually identically. This method is called, with the status reported back from FMS, after every attempt to communicate with the device, and processing falls into one of three cases, depending on the status reported (although the two failure cases are nearly identical).

If the device now being reported OK and it was already OK, there is no change in status, and all that is necessary is to update the m_lastContactTime of the device. (This variable is used to determine when to poll [see runPollDMSTask] and when to declare that a “Communications Timeout” has occurred [see runCheckCommLossTask].) If the status has just become OK, this fact is logged, and the new DMSStatus is persisted and pushed out into the event channel. A request is added to the CommandQueue to poll the device as soon as possible to determine exactly what the status of the sign is. (This is the one exception to the rule that commands on the CommandQueue are processed first-in, first out. The poll command is inserted at the top of the queue so that it is the next command to execute. The DMS cannot easily be polled at this point, because there may be an operation in progress, but in the interest of timeliness, we want to poll ASAP.) Finally, the ArbitrationQueue is notified, so that if there has been a message up on the sign it can notify the controlling TrafficEvent(s) (this is for logging purposes only, it takes no other action).

If the device is now being reported with a failure and the device was already in that failure condition, there is no change in status, and nothing is done. If the status is just now changing, this is logged, and the DMSStatus is persisted and pushed out into the event channel. Finally, the ArbitrationQueue is notified, so that if there has been a message up on the sign it can notify the controlling TrafficEvent(s) (this is for logging purposes only, it takes no other action). Note that if the device has gone into COMM_FAILURE, and it remains in this condition for the timeout period, the CheckCommLossTask’s run method will detect and handle it (see runCheckCommLossTask). Until the timeout period expires, it is assumed that the message is still on the sign, so no further action is taken now. If the device has gone into HARDWARE_FAILURE, FMS is still in contact with it, and changes in status (e.g., loss of a message) can be detected by other means, for instance, by polling (see runPollDMSTask)

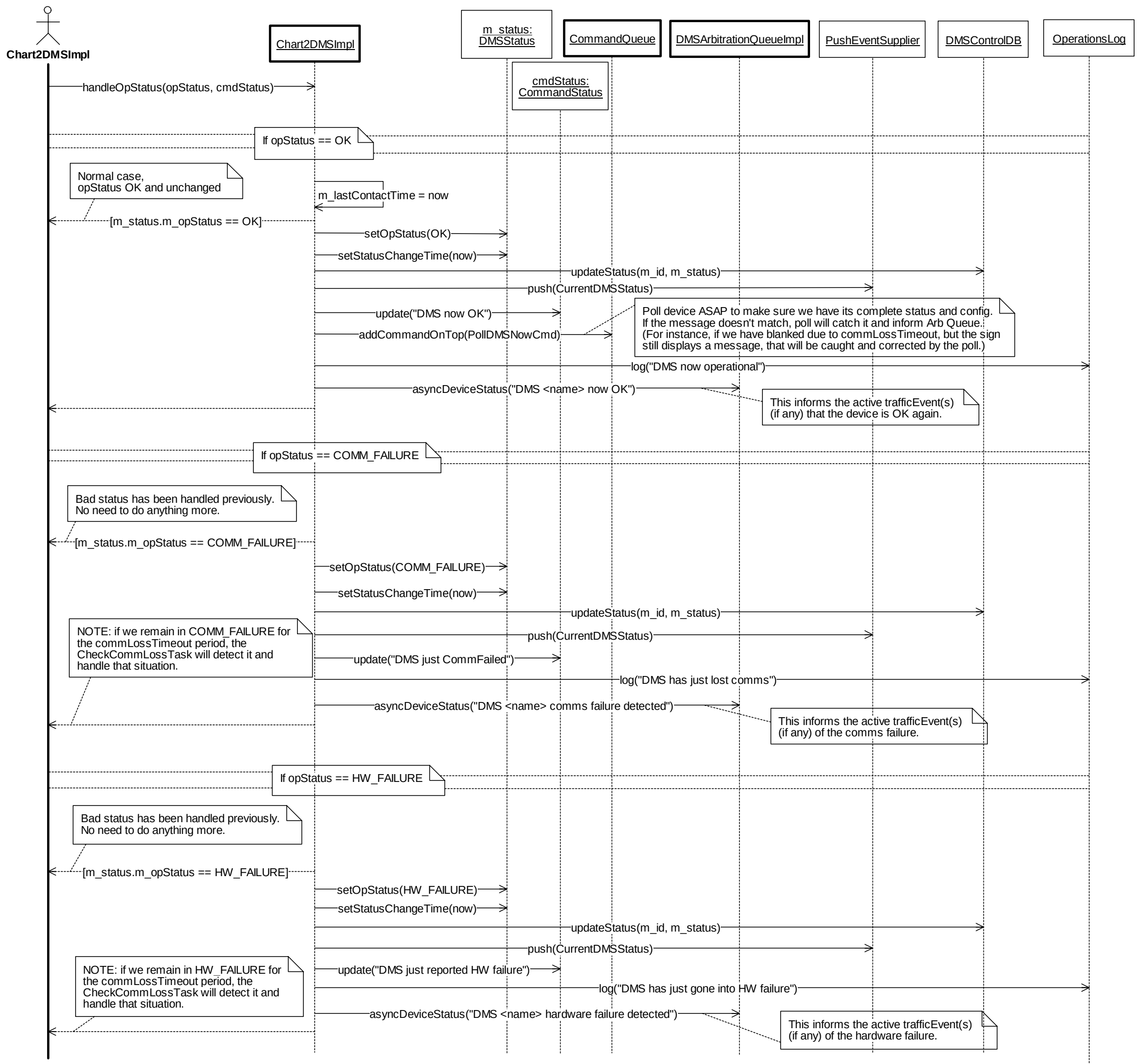

Figure 45. DMSControlModule:HandleOpStatus (Sequence Diagram)

3.6.2.11 DMSControlModule:HasControlledResources (Sequence Diagram)

This Sequence Diagram shows how the CHART2DMSFactoryImpl handles a request to see if an operations center has any controlled resources. The CHART2DMSFactoryImpl simply asks each CHART2DMSImpl for its controlling operations center, and if it matches the OperationsCenter in question, a value of true is immediately returned to the caller. If the CHART2DMSFactoryImpl makes it through its whole list of DMS objects without finding an OperationsCenter match, a value of false is returned.

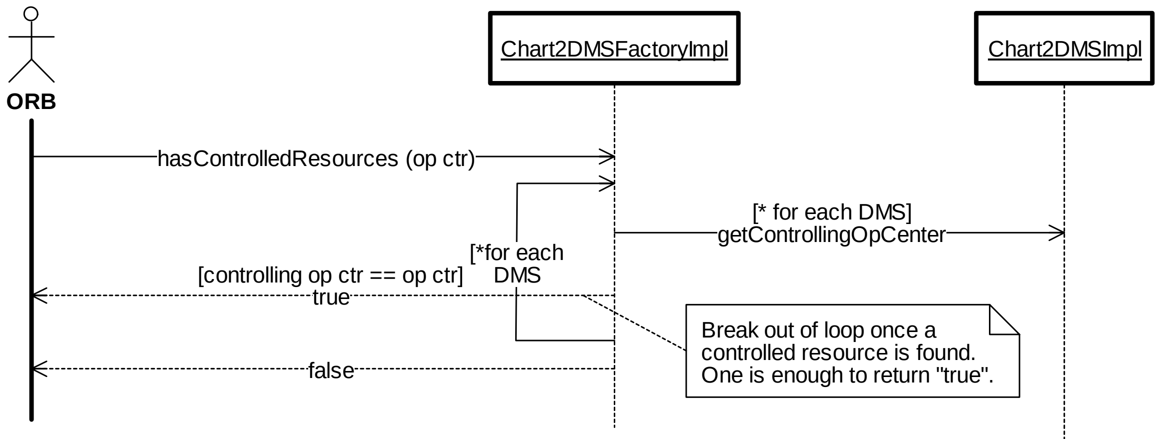

Figure 46. DMSControlModule:HasControlledResources (Sequence Diagram)

3.6.2.12 DMSControlModule:Initialize (Sequence Diagram)

This Sequence Diagram shows how the DMSControlModule is started. This module is created by a service application that will host this module's objects. A ServiceApplication is passed to this module's initialize method and provides access to basic objects needed by this module. This module creates a DMSFactory, which creates the known DMS objects, which have been persisted into the database. The DMSFactory and DMS objects are published via the CORBA Trading Service to make them available for general status updates and as candidates for control (given the proper access rights). In addition to servicing CORBA requests, this service also performs regularly recurring maintenance functions controlled by timer tasks started by this initialize method.

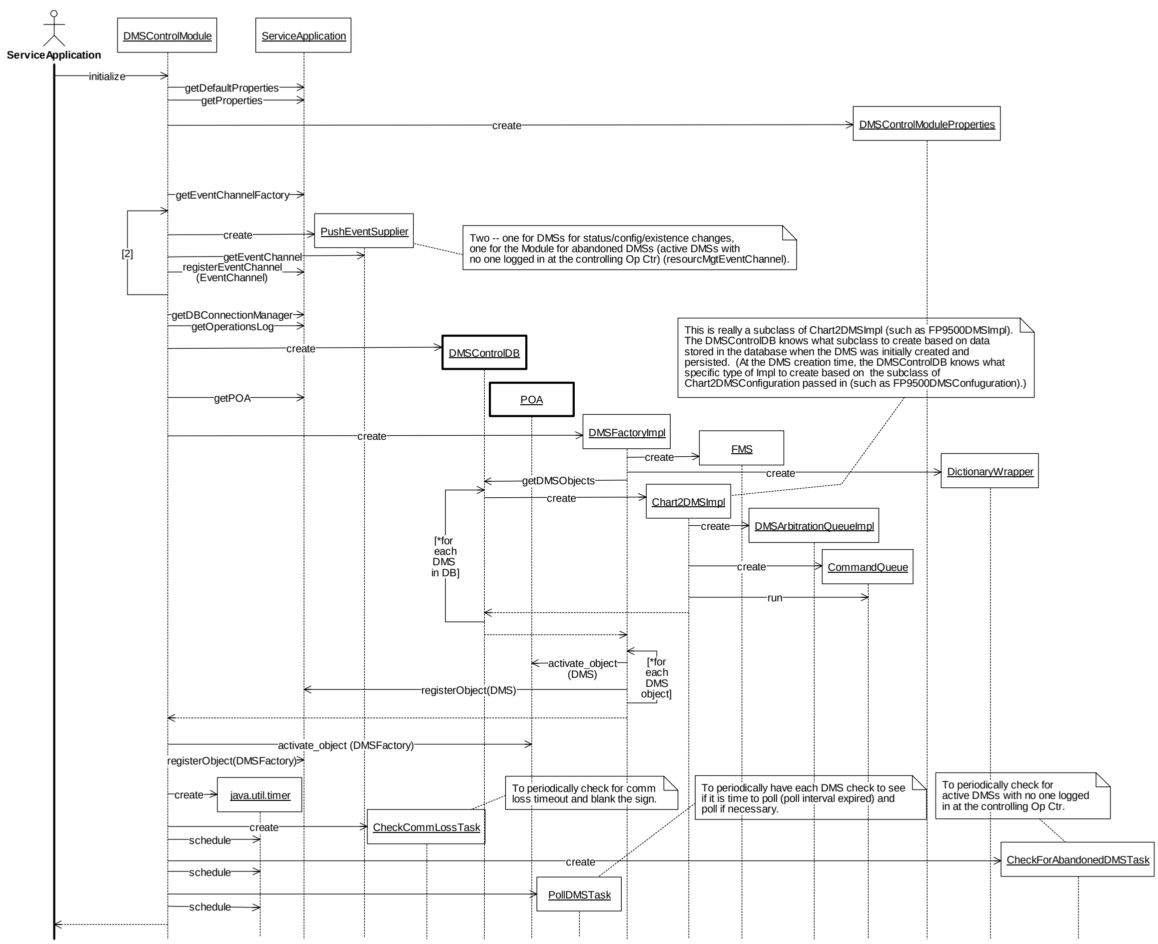

Figure 47. DMSControlModule:Initialize (Sequence Diagram)

3.6.2.13 DMSControlModule:PollNow (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object responds to a request by an operator to immediately poll the device. The DMS must be in maintenance mode and operator must posess proper functional rights. This method creates a PollDMSNowCmd (a QueueableCommand) and adds it to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. Requests to communicate with the sign are processed on a first-come, first-served basis. When the CommandQueue is ready, it executes the PollDMSNowCmd, which calls the pollNowImpl method. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

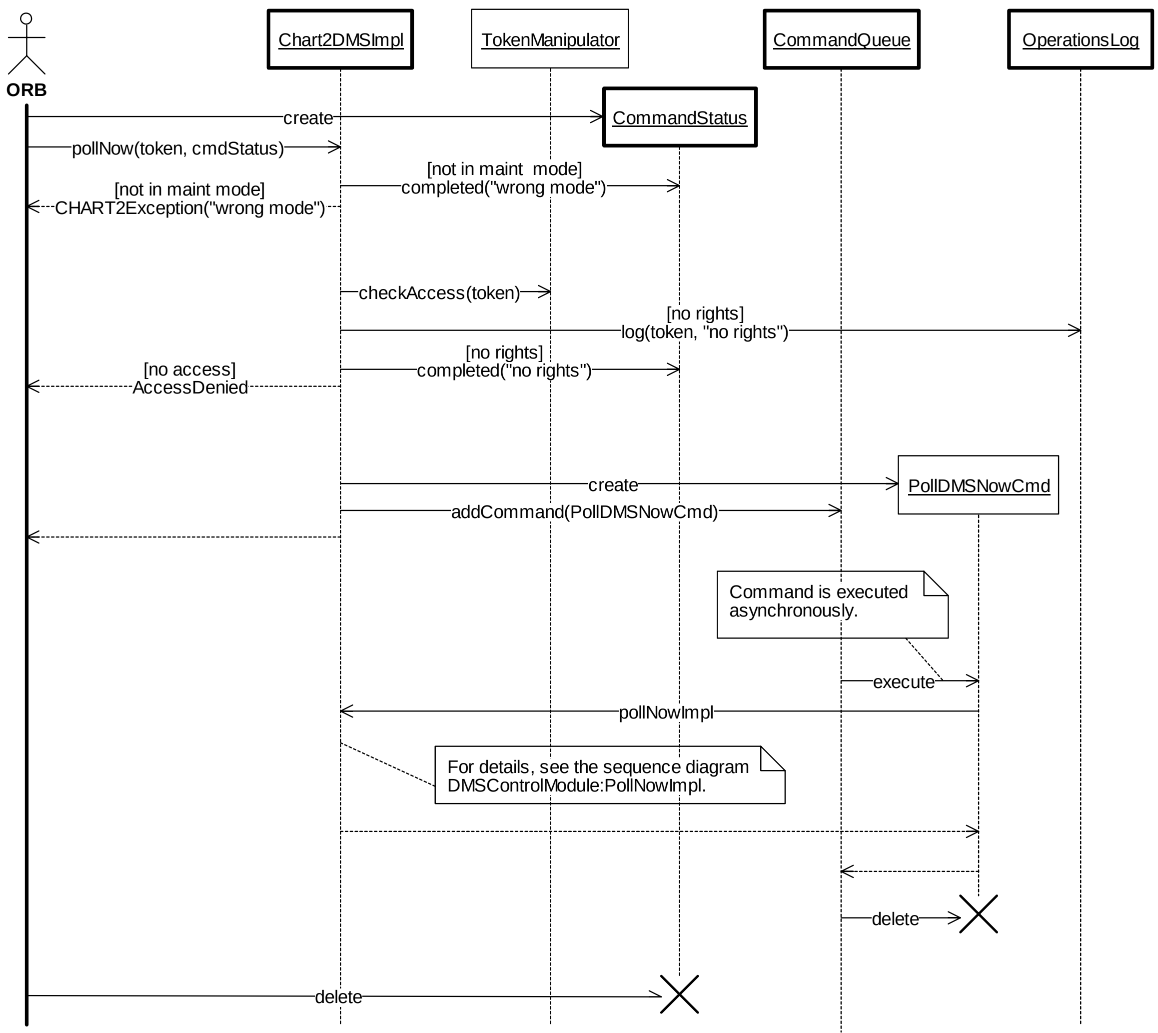

Figure 48. DMSControlModule:PollNow (Sequence Diagram)

3.6.2.14 DMSControlModule:PollNowImpl (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object implements the polling of the DMS device. The poll request could come from the operator (via the pollNow method) or from the automated polling thread within the DMS service itself (PollDMSTask's run method). The pollNowImpl method issues a ForcedPoll request to FMS and calls the method handleOpStatus to detect and handle any changes to the operational status of the sign (OK, comms failure, or hardware failure) reported by FMS during this operation. The status returned is persisted to the database and pushed out as a CurrentDMSStatus event on the event channel. Updates are also written to a CommandStatus object, so that if a user issued this request, he or she can see monitor its progress.

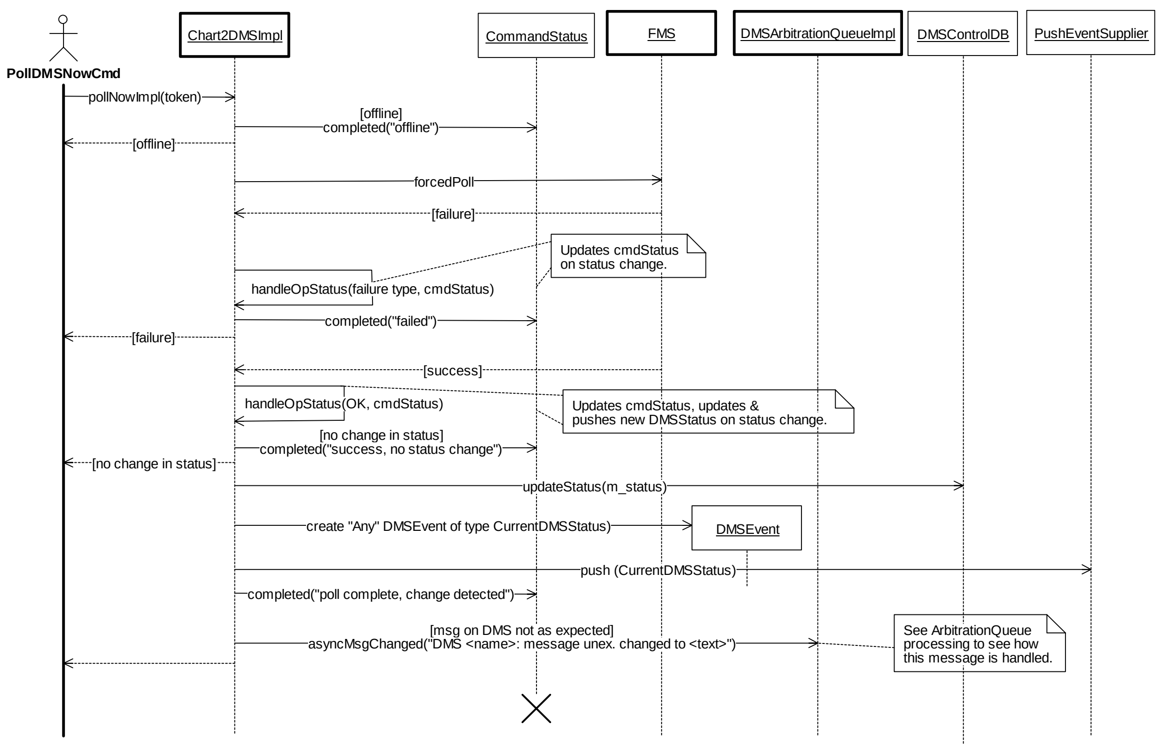

Figure 49. DMSControlModule:PollNowImpl (Sequence Diagram)

3.6.2.15 DMSControlModule:PutDMSInMaintMode (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object responds to a request by a user to go into maintenance mode. The requesting operator must have proper functional rights, and if there is a message on the sign from another operations center, the user must have override authority. And of course the sign must not be in maintenance mode already, otherwise the request is redundant. The ArbitrationQueue is interrupted, so that it will stop attempting to modify the sign (as it does in online mode). A PutDMSInMaintModeCmd (a QueueableCommand) is created and added to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. When the CommandQueue is ready, it executes the PutDMSInMaintModeCmd, which calls the putInMaintModeImpl method, also shown on this diagram. The putInMaintModeImpl method double checks to make sure it is not already in maintenance mode (from some other queued command). Assuming no problems, the method blankSignNow is called to request FMS to actually blank the sign, update the database, and handle any status change, and push a CurrentDMSStatus event into the event channel, so that any user can immediately see that the sign is now blank. Regardless of whether blankSignNow works, the method continues on, since the sign may likely be non-functional when it is put in maintenance mode. The DMSStatus is updated to show that the sign is in maintenance mode, it is persisted to the database, and it is pushed into the event channel. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

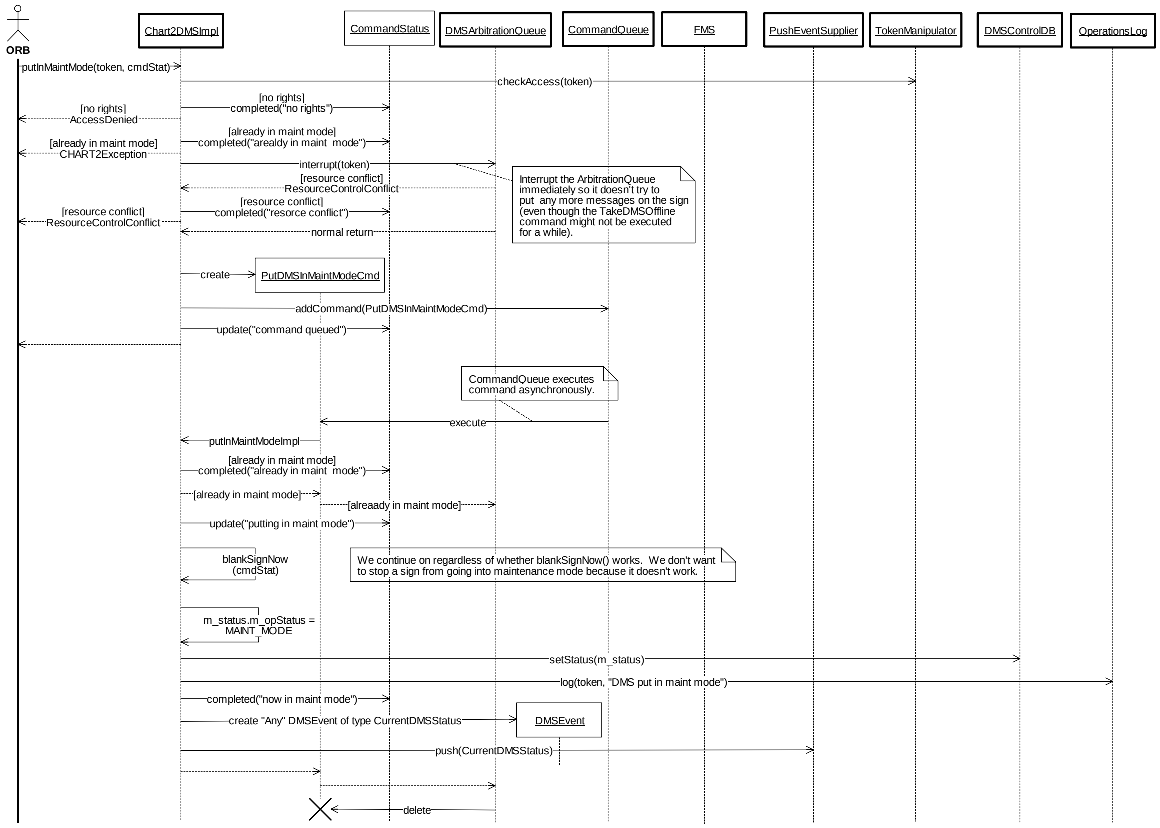

Figure 50. DMSControlModule:PutDMSInMaintMode (Sequence Diagram)

3.6.2.16 DMSControlModule:PutDMSOnline (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object responds to a request by a user to go online. The requesting operator must have proper functional rights, and if there is a (maintenance mode) message on the sign from another operations center, the user must have override authority. And of course the sign must not online already, otherwise the request is redundant. A PutDMSOnlineCmd (a QueueableCommand) is created and added to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. When the CommandQueue is ready, it executes the PutDMSOnlineCmd, which calls the putOnlineImpl method, also shown on this diagram. The putOnlineImpl method double checks to make sure it is not already online (from some other queued command). Assuming no problems, the method blankSignNow is called to request FMS to actually blank the sign, update the database, and handle any status change, and push a CurrentDMSStatus event into the event. If blankSignNow does not work, the sign cannot be brought online, and the method ends. The DMSStatus is updated to show that the sign is online, it is persisted to the database, and it is pushed into the event channel. Finally the ArbitrationQueue is resumed, so that it can evaluate its queue and determine if it has a message to display on the sign. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

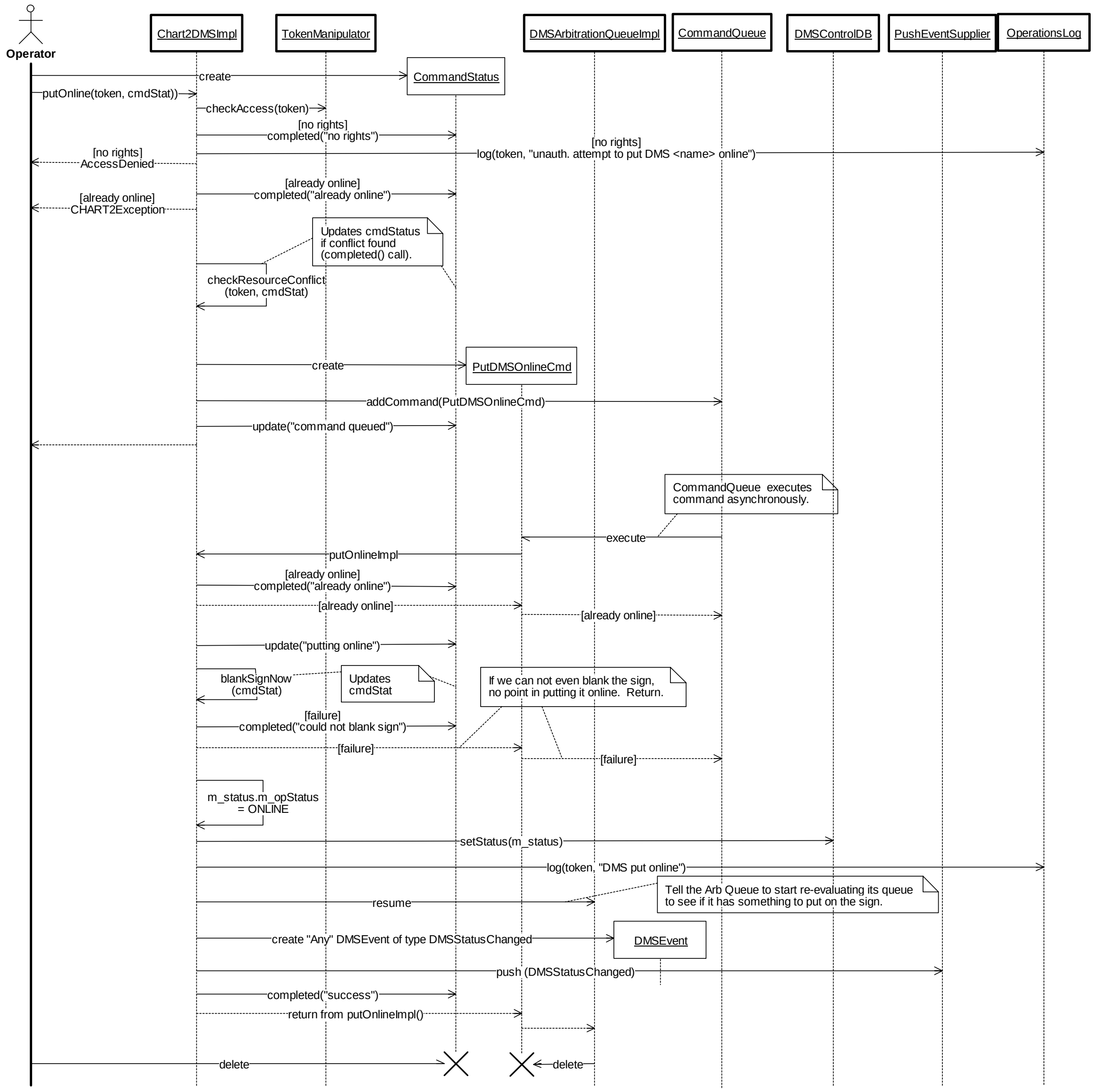

Figure 51. DMSControlModule:PutDMSOnline (Sequence Diagram)

3.6.2.17 DMSControlModule:RemoveDMS (Sequence Diagram)

This Sequence Diagram shows how the DMSFactoryImpl removes a DMS from the system on behalf of an operator. A DMS must be offline to be removed, and the requesting operator must posess the proper functional rights. The DMSFactory remove the reference to the DMSImpl from its internal list of DMSs, remove the DMSImpl and its associated information from the database removes it from the FMS subsystem, and withdraws the DMS's offer from the trading service. A DMSDeletedEvent is then pushed into the event channel.

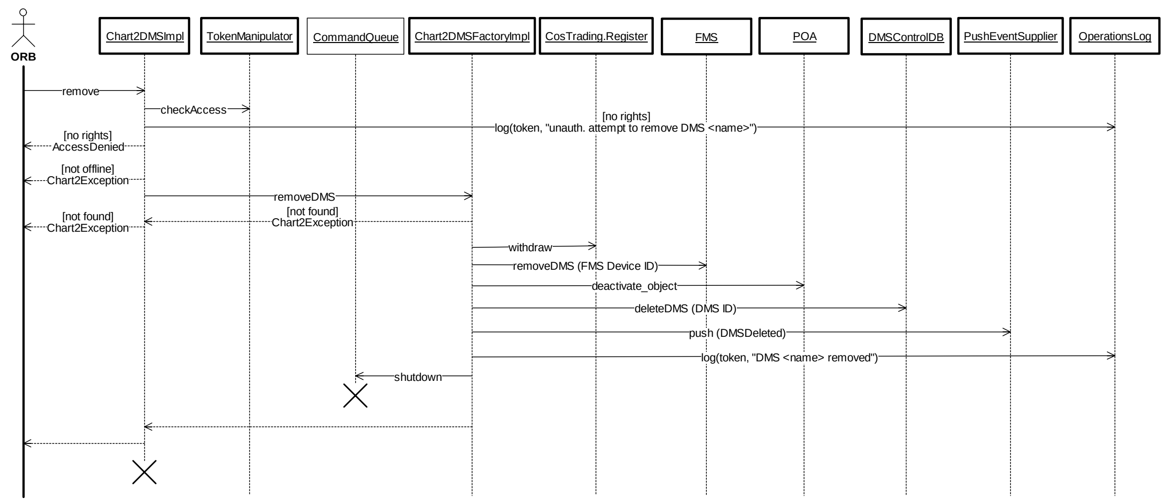

Figure 52. DMSControlModule:RemoveDMS (Sequence Diagram)

3.6.2.17.1 DMSControlModule:ResetController (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl responds to a request to reset a DMS. The DMS must be in maintenance mode, the requesting operator must have proper functional rights, and if there is a (maintenance mode) message on the sign from another operations center, the user must have override authority. This method creates a ResetDMSCmd (a QueueableCommand) and adds it to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. Requests to communicate with the sign are processed on a first-come, first-served basis. When the CommandQueue is ready, it executes the ResetDMSCmd, which calls the resetControllerImpl method, also shown on this diagram. When the resetControllerImpl method runs, it checks that the DMS is still in maintenance mode (a previously queued command could have changed it), and that there is no resource conflict (a previously queued command could have written a message from an operator at another operations center). Assuming no problems, the method blankSignNow is called to request FMS to actually change the sign, update the database, and handle any status change, and push a CurrentDMSStatus event into the event channel, so that any user (with rights) can immediately see that the sign is now blank. Then the FMS is requested to reset the device with the FMS's resetController method. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

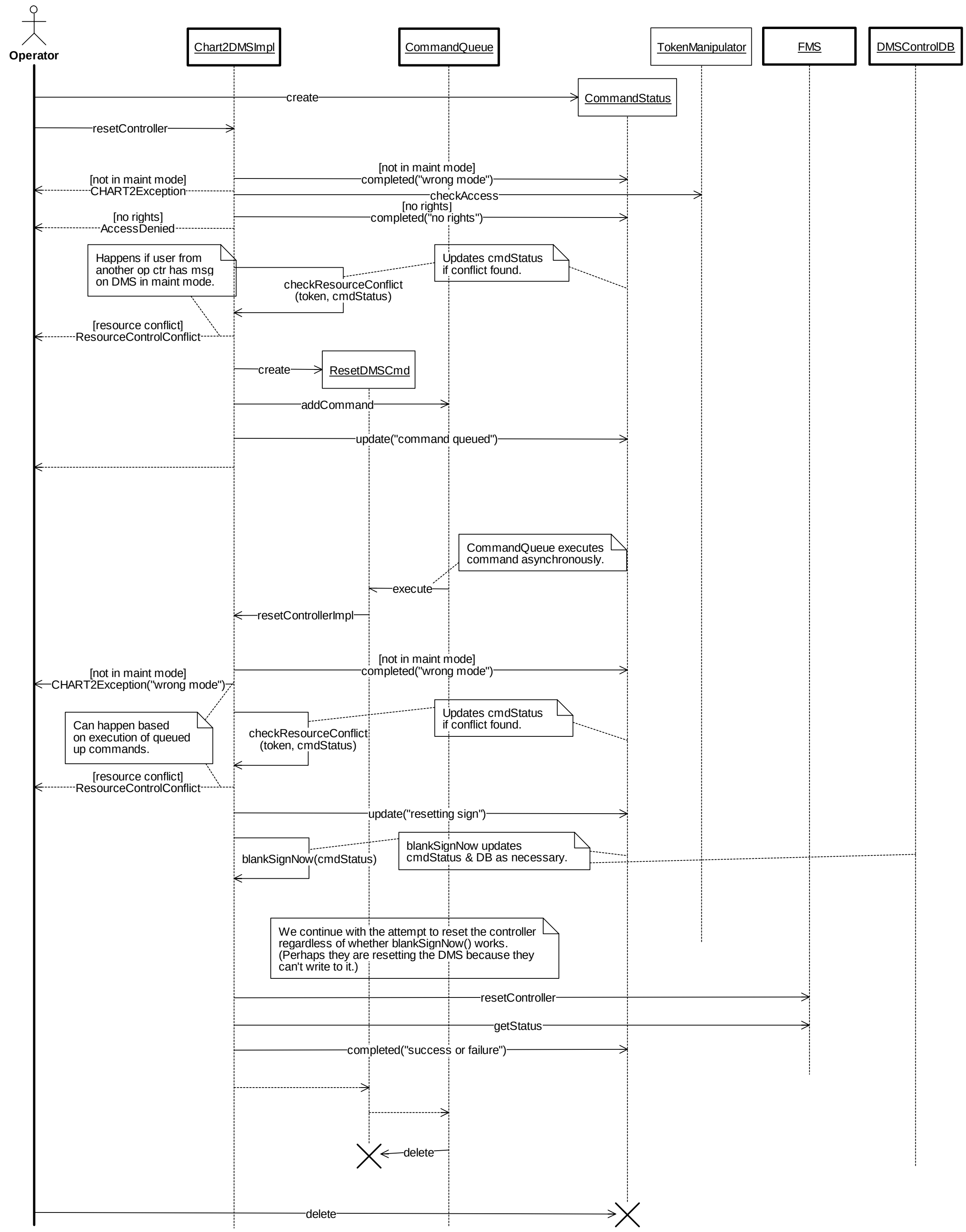

Figure 53. DMSControlModule:ResetController (Sequence Diagram)

3.6.2.18 DMSControlModule:RunCheckCommLossTask (Sequence Diagram)

This Sequence Diagram shows how the CheckCommLossTask object executes its task when directed to run by the Java timer object. The run method of CheckCommLossTask calls the checkCommLoss method of CHART2DMSFactoryImpl, which calls checkCommLoss on each DMS. Each CHART2DMSImpl object immediately returns if its m_lastContactTime variable indicates that it has had some (any) communication with the device within the Comm Loss Timeout period. If the timeout has been exceeded and there was a message on the sign, the CHART2DMSStatus is updated to reflect a blank message and no controlling operations center, this fact is logged, and the new status is persisted and pushed into the event channel. (If the timeout has been exceeded, but there is no message on the sign, there is nothing to do and no one to notify. The COMM_FAILURE status has already been detected, on the first failed poll if nothing else.)

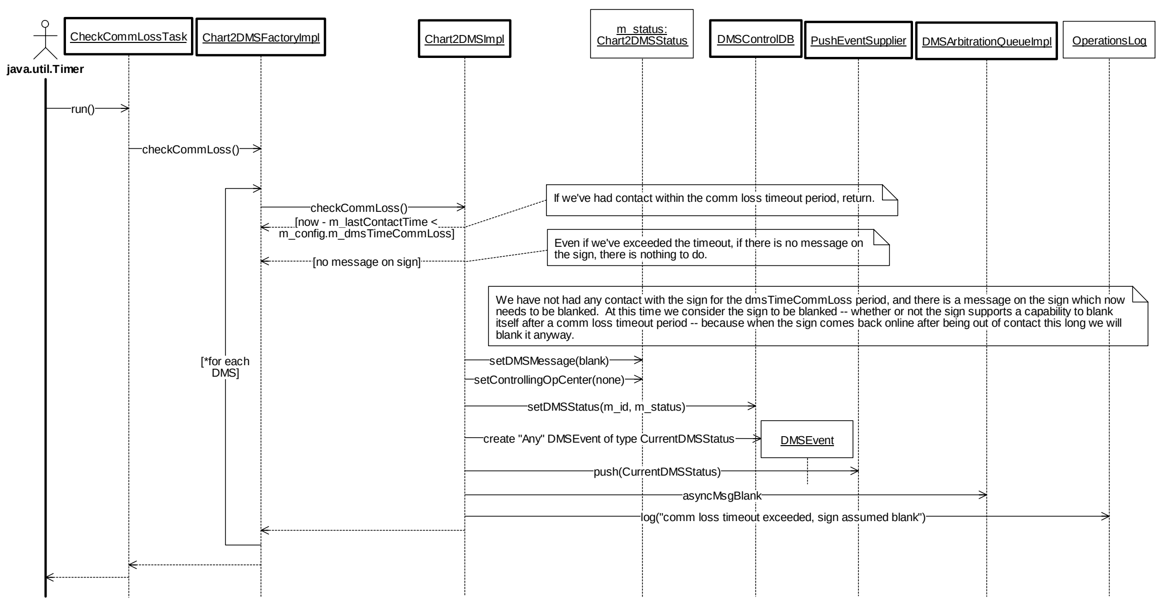

Figure 54. DMSControlModule:RunCheckCommLossTask (Sequence Diagram)

3.6.2.19 DMSControlModule:RunCheckForAbandonedDMSTask (Sequence Diagram)

This Sequence Diagram shows how the CheckForAbandonedDMSTask object executes its task when directed to run by the Java timer object. The run method of CheckForAbandonedDMSTask gets the controlling op center of each DMS and builds a list of OperationsCenter objects with control one or more signs. Each OperationsCenter is then queried for the number of users logged in. If the number of users at an OperationsCenter is zero, this fact is logged and an UnhandledControlledResources event is pushed into the event channel.

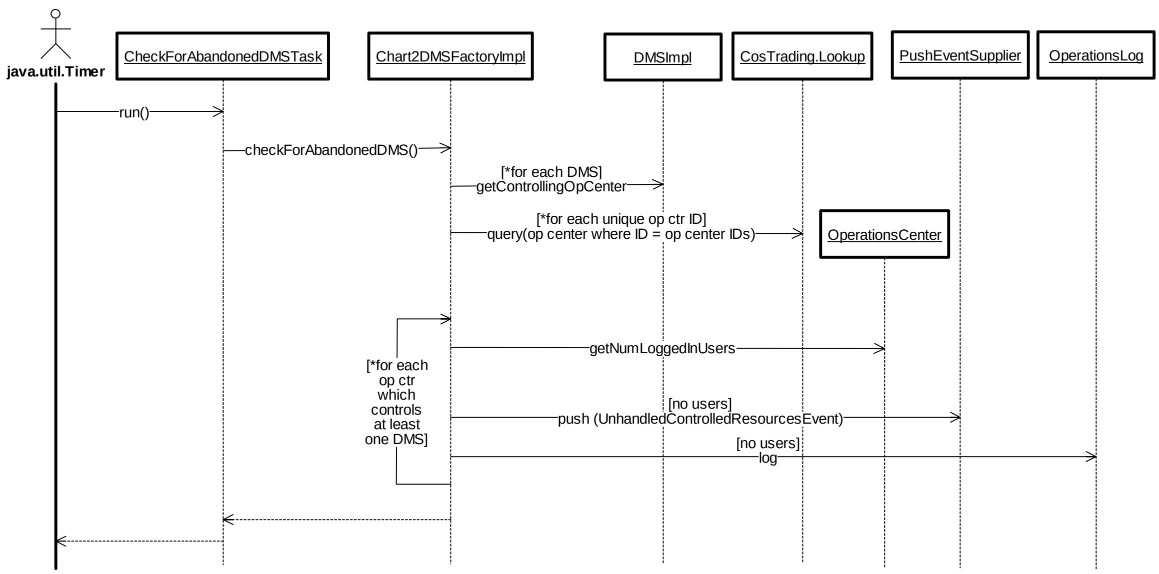

Figure 55. DMSControlModule:RunCheckForAbandonedDMSTask (Sequence Diagram)

3.6.2.20 DMSControlModule:RunPollDMSTask (Sequence Diagram)

This Sequence Diagram shows how the PollDMSTask object executes its task when directed to run by the Java timer object. The run method of PollDMSTask calls the pollDMSes method of CHART2DMSFactoryImpl, which calls pollIfNecessary on each DMS. Each CHART2DMSImpl object immediately returns if its m_lastContactTime variable indicates that it has had some (any) communication with the device within the poll interval period. If it has been longer than the poll interval since the last communcation with the device, this method creates a PollDMSNowCmd (a QueueableCommand) and adds it to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. Requests to communicate with the sign are processed on a first-come, first-served basis. Most likely, the CommandQueue is empty (which is why a need to poll is indicated), but any communication with the device will have the desired effect. If there are one or more requests to communicate with the device on the queue ahead of this PollDMSNowCmd, that is acceptable, too. When the CommandQueue is ready, it executes the PollDMSNowCmd, which calls the pollNowImpl method.

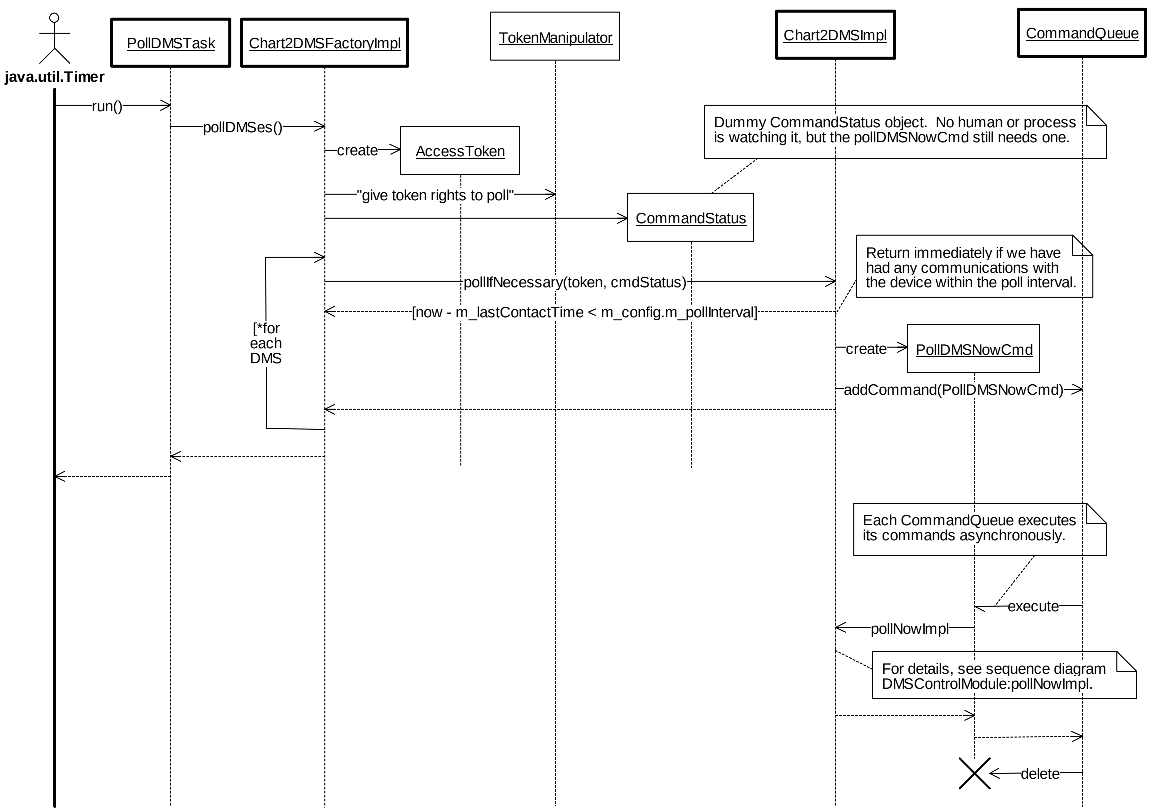

Figure 56. DMSControlModule:RunPollDMSTask (Sequence Diagram)

3.6.2.21 DMSControlModule:SetConfiguration (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl responds to a request to change the configuration of a DMS. The DMS must be in maintenance mode, the requesting operator must have proper functional rights, and if there is a (maintenance mode) message on the sign from another operations center, the user must have override authority. This method creates a SetDMSConfigCmd (a QueueableCommand) and adds it to the DMS's CommandQueue. The CommandQueue is required since some configuration changes require field communications to the sign, and field communications are relatively slow and can queue up. Requests to communicate with the sign are processed on a first-come, first-served basis. When the CommandQueue is ready, it executes the SetDMSConfigCmd, which calls the setConfigurationImpl method, also shown on this diagram. When the setConfigurationImpl method runs, it checks that the DMS is still in maintenance mode (a previously queued command could have changed it), and that there is no resource conflict (a previously queued command could have written a message from an operator at another operations center). Assuming no problems, the CHART2DMSConfiguration is locked down, and all parameters that need to change are changed. If any of these parameter changes require communications to the sign (e.g., setting the Comm Loss Timeout in an FP9500), FMS is requested to make the specified change(s). The method handleOpStatus handles and responds to any changes to the operational status of the sign (OK, comms failure, or hardware failure) reported by FMS during this operation. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

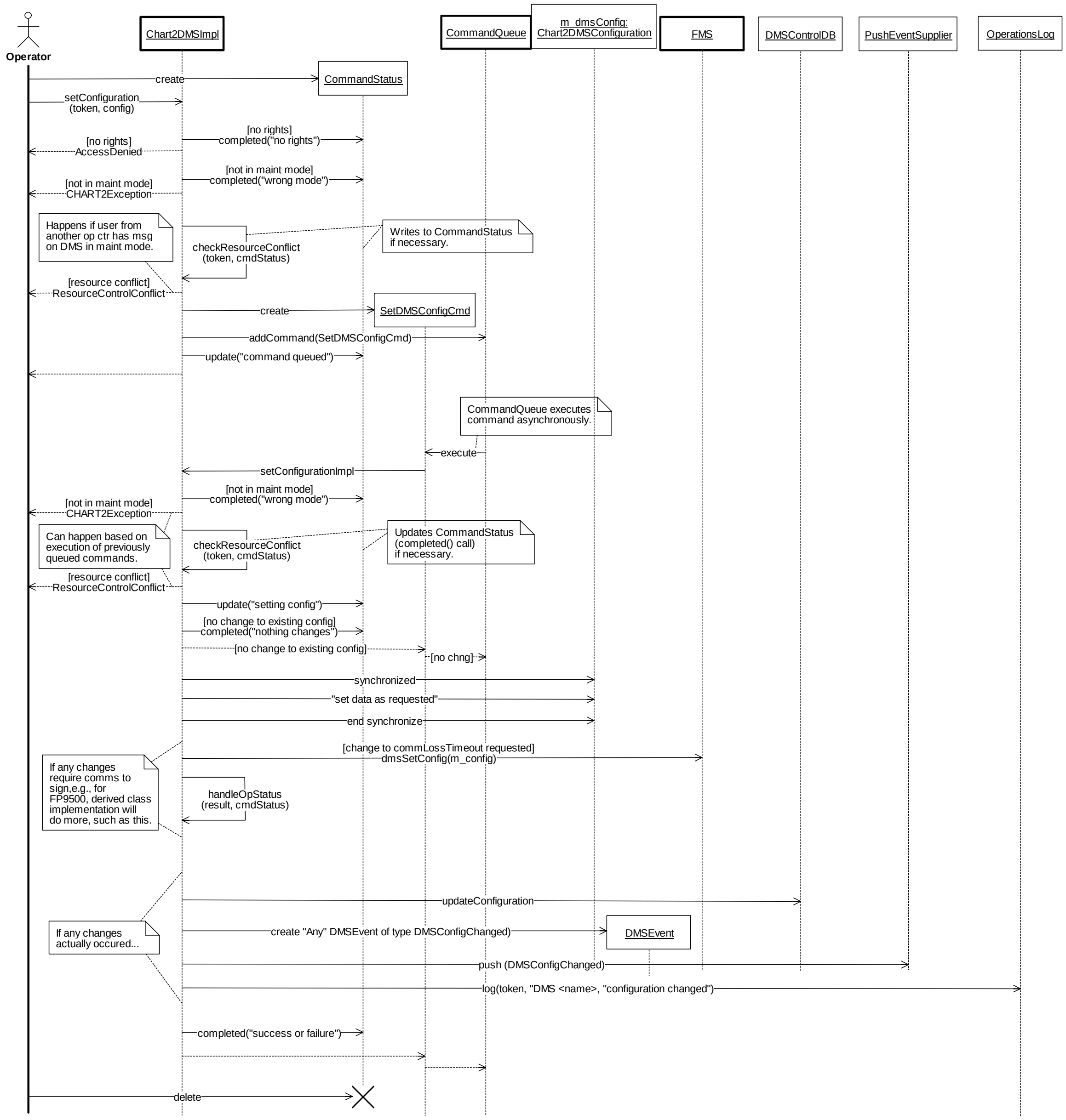

Figure 57. DMSControlModule:SetConfiguration (Sequence Diagram)

3.6.2.22 DMSControlModule:SetMessage (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object processes a request to change its message in maintenance mode. (For setting messages online, see SetMessageFromQueue.) The DMS must be in maintenance mode, and the requesting operator must have proper functional rights. This method asks the message to validate itself one last time (for banned words, and to ensure that the beacons are not set on with an empty message). Then a SetDMSMessageCmd (a QueueableCommand) is created and added to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. Requests to communicate with the sign are processed on a first-come, first-served basis. When the CommandQueue is ready, it executes the SetDMSMessageCmd, which calls the setMessageImpl method. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

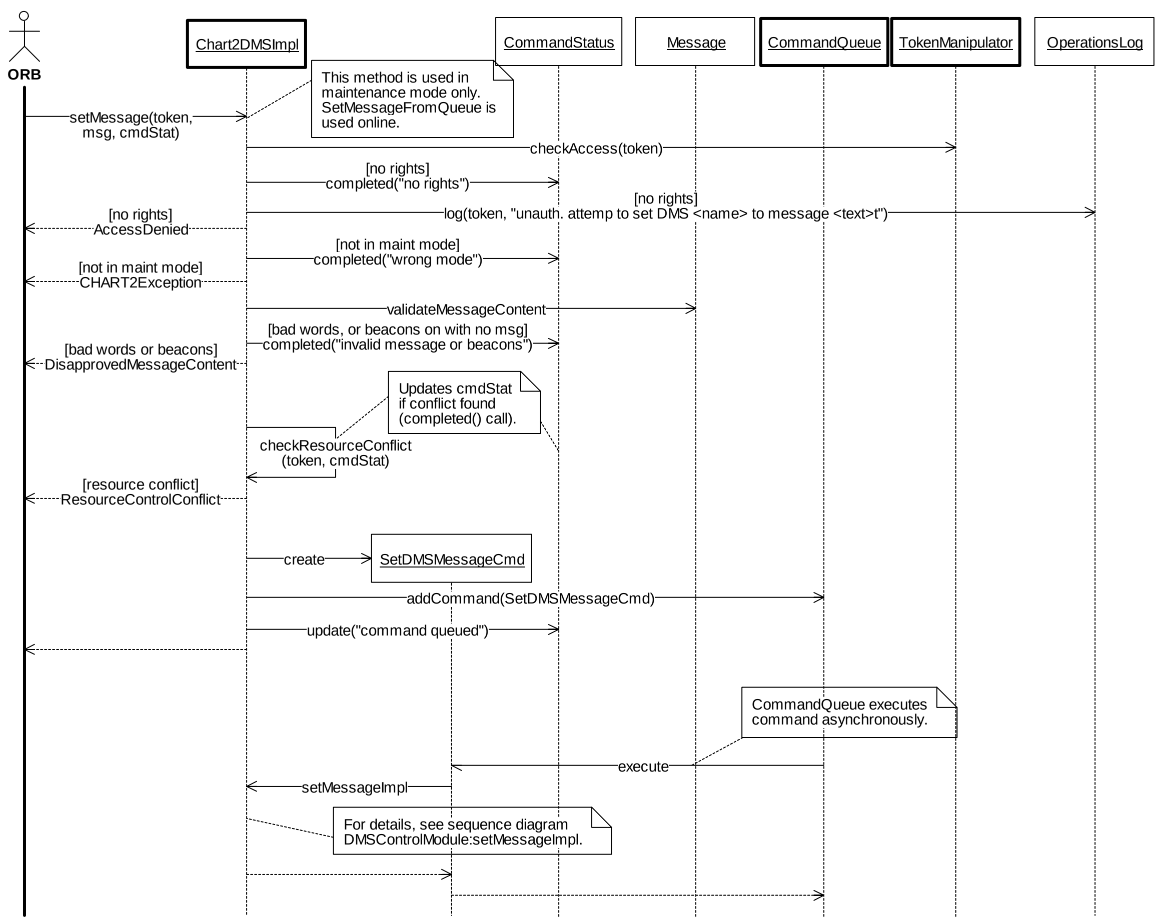

Figure 58. DMSControlModule:SetMessage (Sequence Diagram)

3.6.2.23 DMSControlModule:SetMessageFromQueue (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object processes a request to change its message while it is online. (For setting messages in maintenance mode, see setMessage.) This thread is actually initiated in the ArbitrationQueue's AddEntry method. The ArbitrationQueue's evaluateQueue method calls this method. The DMS must still be online. The operator's functional rights have already been validated. This method creates a SetDMSMessageFromQueueCmd (a QueueableCommand) and adds it to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. When the CommandQueue is ready, it executes the SetDMSMessageFromQueueCmd, which calls the setMessageFromQueueImpl method. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

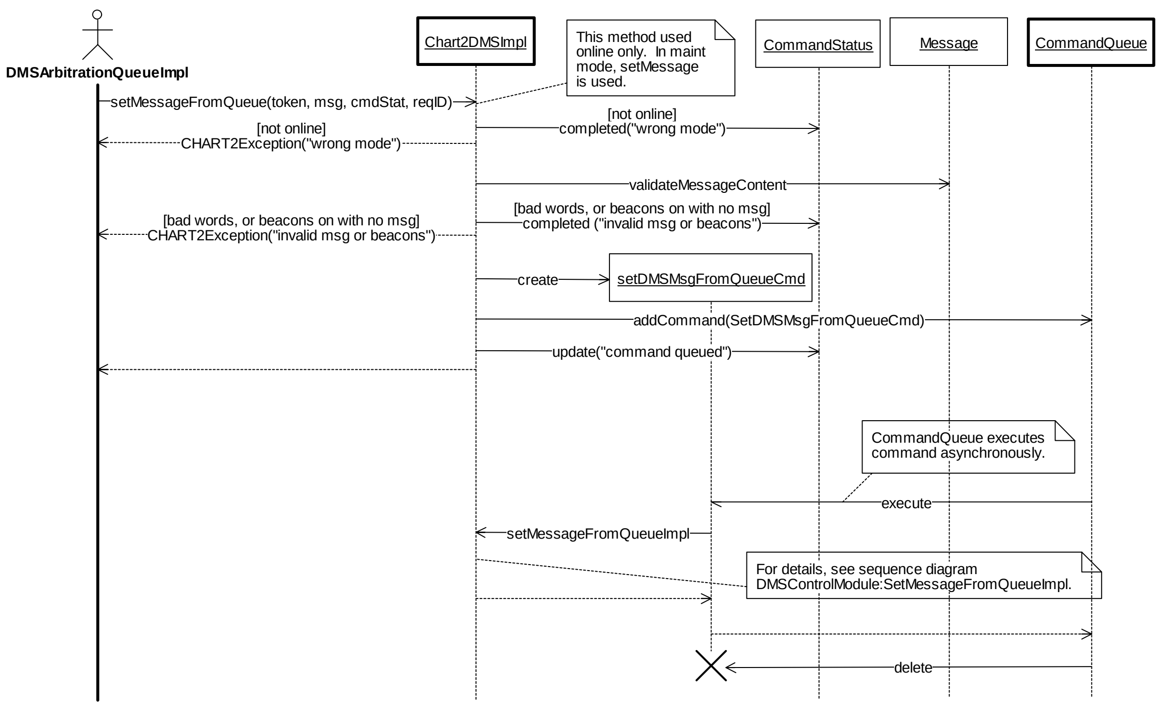

Figure 59. DMSControlModule:SetMessageFromQueue (Sequence Diagram)

3.6.2.24 DMSControlModule:SetMessageFromQueueImpl (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object executes a command to change its message while it is online. (The analogous method in online mode is setMessageImpl.) A request to set the message has already been received and pre-processed by the setMessageFromQueue method. When the setMessageFromQueueImpl method runs, it checks that the DMS is still online (a previously queued command could have changed it), that the user has rights, and that there is no resource conflict (a previously queued command could have written a message from an operator at another operations center). Assuming no problems, FMS is reqeuested to change the sign. If it succeeds, the controlling operations center is updated as necessary, and the database is updated with the new information. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user. A CurrentDMSStatus event is pushed into the event channel, so that any user (with rights) can immediately see the new content of the sign. The method handleOpStatus handles and responds to any changes to the operational status of the sign (OK, comms failure, or hardware failure) reported by FMS during this operation. The ArbitrationQueue is informed of the result of this operation via the requestFailed or requestSucceeded method (at which time the ArbitrationQueue may re-evaluate its own queue and request another change to the sign).

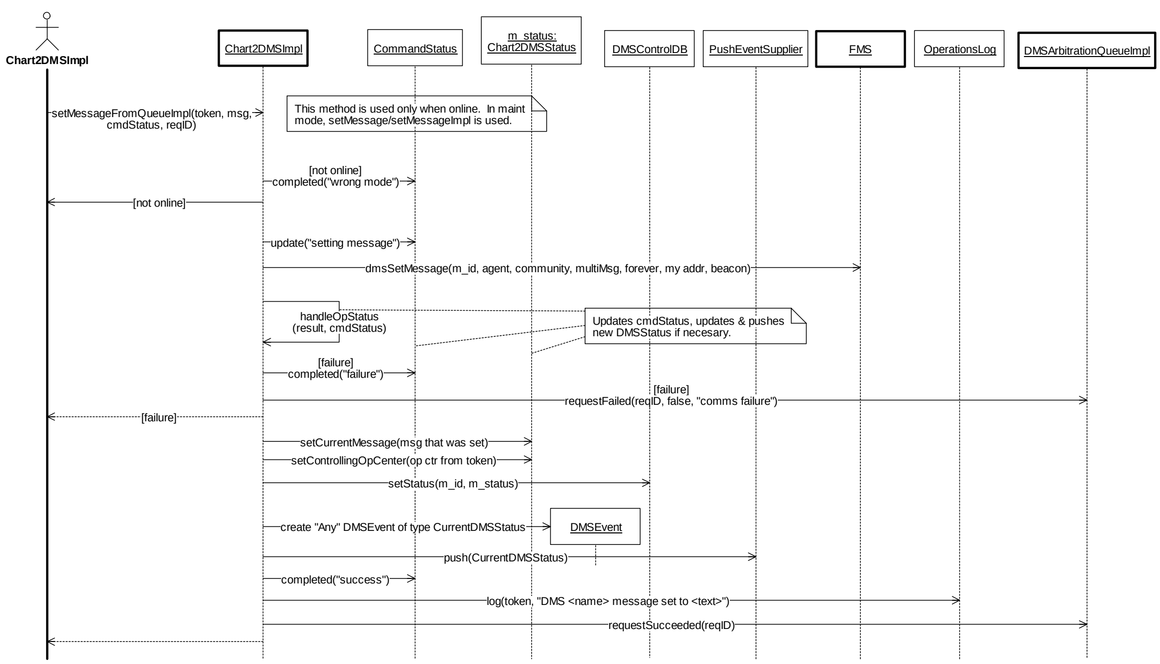

Figure 60. DMSControlModule:SetMessageFromQueueImpl (Sequence Diagram)

3.6.2.25 DMSControlModule:SetMessageImpl (Sequence Diagram)

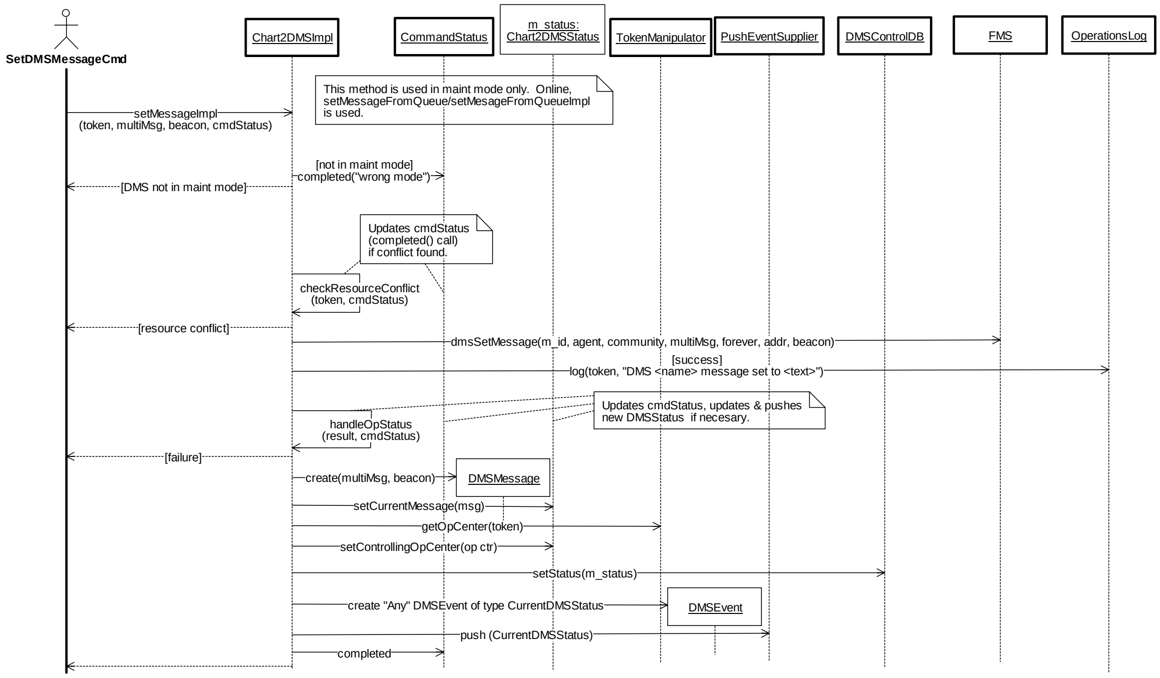

Figure 61. DMSControlModule:SetMessageImpl (Sequence Diagram)

3.6.2.26 DMSControlModule:Shutdown (Sequence Diagram)

This Sequence Diagram shows how the DMSControlModule is terminated. The DMSControlModule is shut down by the ServiceApplication that started it. When told to shut down, the DMSControlModule disconnects the DMSFactory from the ORB, withdraws its offer from the trader, and shuts down the object. When the DMSFactory is shut down, it withdraws the offers of each DMS and disconnects each DMS from the ORB. No information needs to be persisted to the database during shutdown, as information is written to the database as it is updated.

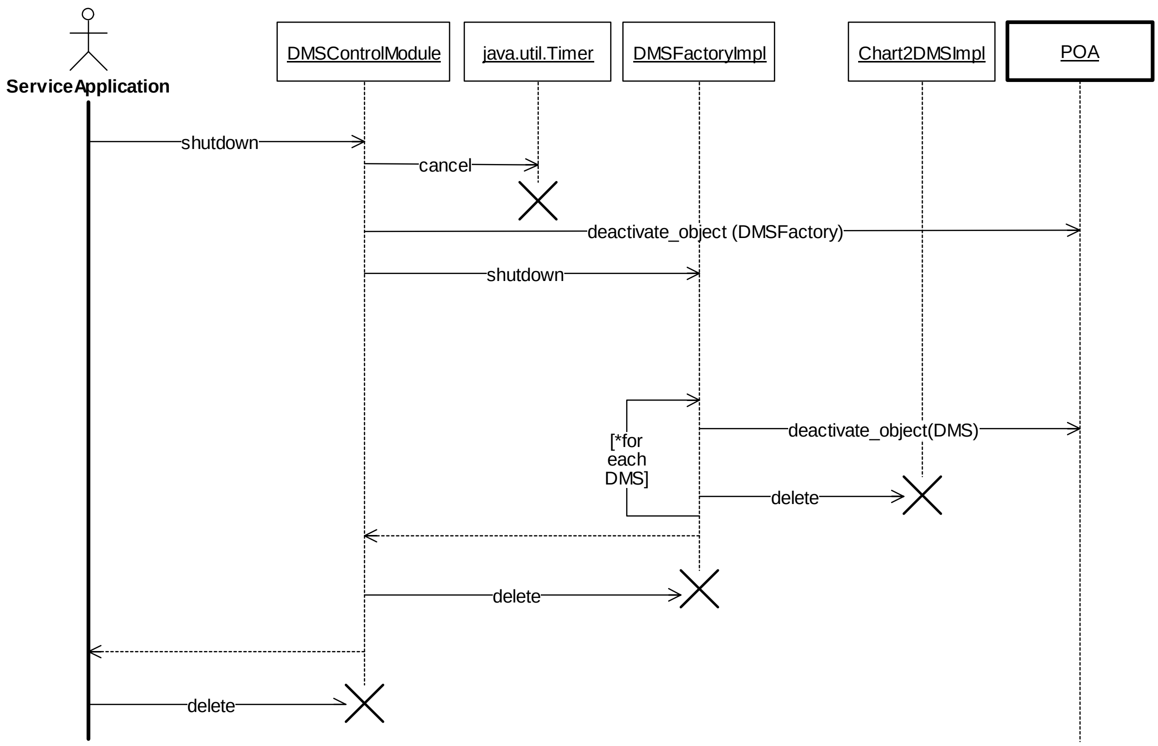

Figure 62. DMSControlModule:Shutdown (Sequence Diagram)

3.6.2.27 DMSControlModule:TakeDMSOffline (Sequence Diagram)

This Sequence Diagram shows how a CHART2DMSImpl object responds to a request by a user to go offline. The requesting operator must have proper functional rights, and if there is a message on the sign from another operations center, the user must have override authority. And of course the sign must not offline already, otherwise the request is redundant. The ArbitrationQueue is interrupted, so that it will stop attempting to modify the sign (as it does in online mode). A TakeDMSOfflineCmd (a QueueableCommand) is created and added to the DMS's CommandQueue. The CommandQueue is required since field communications to the sign are relatively slow and can queue up. When the CommandQueue is ready, it executes the TakeDMSOfflineCmd, which calls the takeOfflineImpl method, also shown on this diagram. The takeOfflineImpl method double checks to make sure it is not already offline (from some other queued command). Assuming no problems, the method blankSignNow is called to request FMS to actually blank the sign, update the database, and handle any status change, and push a CurrentDMSStatus event into the event channel, so that any user (with rights) can immediately see that the sign is now blank. Regardless of whether blankSignNow works, the method continues on, since the sign may likely be non-functional when it is taken offline. The DMSStatus is updated to show that the sign is offline, it is persisted to the database, and it is pushed into the event channel. The requesting user is kept abreast of progress of the request all the while, via a CommandStatus object viewable by the user.

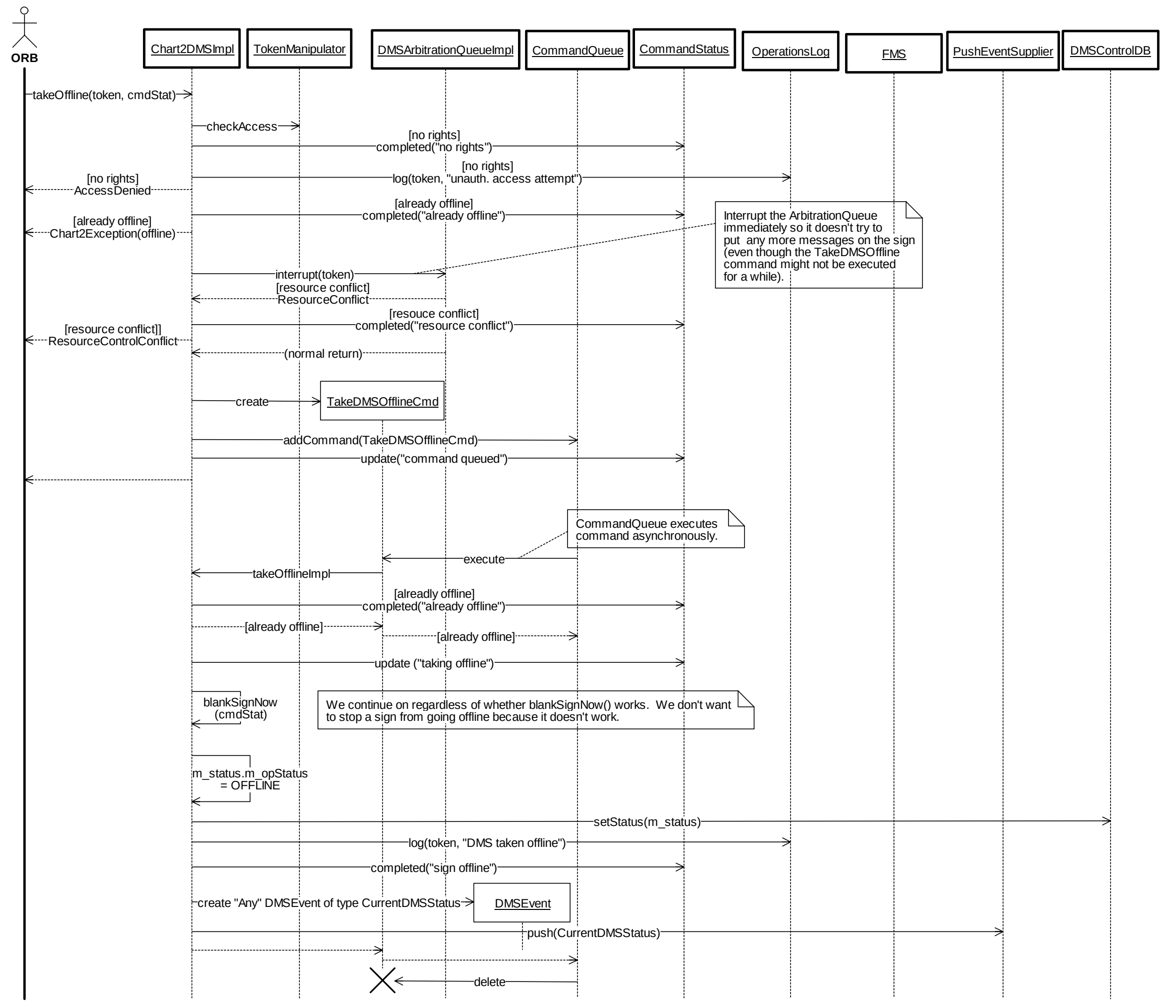

Figure 63. DMSControlModule:TakeDMSOffline (Sequence Diagram)

3.7 DMSUtility

3.7.1 Classes

3.7.1.1 DMSUtility (Class Diagram)

This Class Diagram shows classes related to the DMS that are used by both the GUI and the DMS service. Most of these classes are implementations of value type classes defined in the system interfaces (IDL).

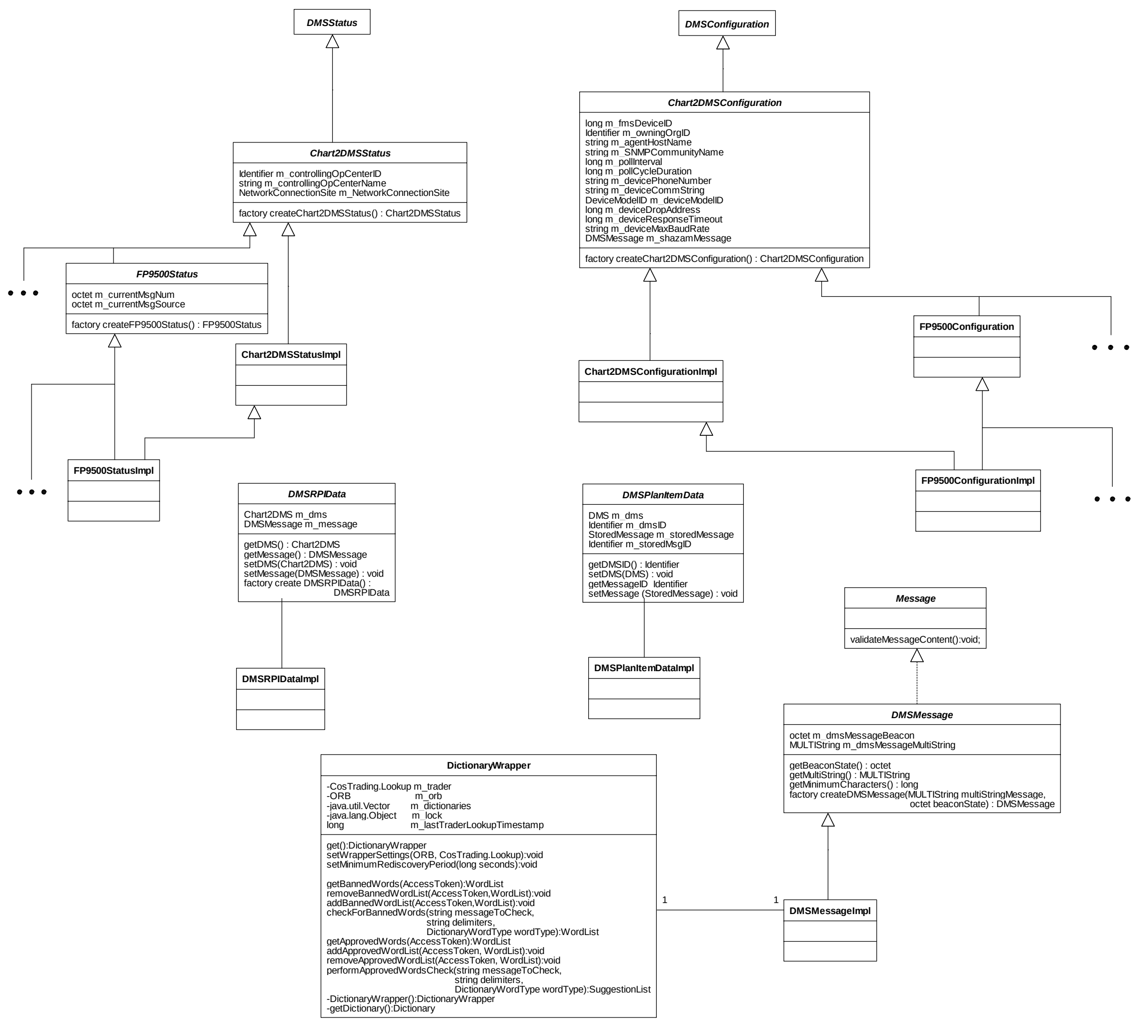

Figure 64. DMSUtility (Class Diagram)

3.7.1.1.1 CHART2DMSConfiguration (Class)

The CHART2DMSConfiguration class is an abstract class which extends the DMSConfiguration class to provide configuration information specific to CHART II processing. Such information includes how to contact the sign under CHART II software control, the default SHAZAM message for using the sign as a HAR Notifier, and the owning organization. Such data extends beyond what would be industry-standard configuration information for a DMS.

3.7.1.1.2 CHART2DMSConfigurationImpl (Class)

The CHART2DMSConfigurationImpl class provides an implementation for the abstract CHART2DMSConfiguration class. It implements get and set methods to access and modify values of the configuration of a DMS. The configuration information stored here is normally fairly static: things like the size of the sign in characters and pixels, its name and location, and how to contact the sign (as opposed to dynamic information like the current message on the sign, which is stored in an analogous Status object).

3.7.1.1.3 CHART2DMSStatus (Class)

The CHART2DMSStatus class is an abstract class that extends the DMSStatus class to provide status information specific to CHART II processing, such as information on the controlling operations center for the sign. This data extends beyond what would be industry-standard status information for a DMS.

3.7.1.1.4 CHART2DMSStatusImpl (Class)

The CHART2DMSStatusImpl class provides an implementation for the abstract CHART2DMSStatus class. It implements get and set methods to access and modify values of the status of a DMS. The status information stored here is relatively dynamic: things like the current message on the sign, its beacon state, its current operational mode (online, offline, maintenance mode), and current operational status (OK, COMM_FAILURE, or HARDWARE_FAILURE) and controlling operations center. (More static information about the sign, such as its size and location, is stored in an analogous Configuration object.)

3.7.1.1.5 DictionaryWrapper (Class)

This singleton class provides a wrapper for the system dictionary that provides automatic location of the dictionary and automatic re-discovery should the dictionary reference return an error. This class also allows for built-in fault tolerence by automatically failing over to a “working” dictionary without the user of this class being aware that this being done. In addition, this class defers the discovery of the Dictionary until its first use, thus eliminating a start-up dependency for modules that use the dictionary.

This class delegates all of its method calls to the system dictionary using its currently known good reference to the system dictionary. If the current reference returns a CORBA failure in the delegated call, this class automatically switches to another reference. When there are no good references (as is true the first time the object is used), this class issues a trader query to (re)discover the published Dictionary objects in the system. During a method call, the trader will be queried at most one time and under normal circumstances (other than the first use) the trader will not be queried at all.

3.7.1.1.6 DMSMessage (Class)

The DMSMessage class is an abstract class that describes a message for a DMS. It consists of two elements: a MULTI-formatted message and beacon state information (whether the message requires that the beacons be on). The DMSMessage is contained within a DMSStatus object, used to communicate the current message on a sign, and is stored within a DMSRPIData object, used to specify the message that should be on a sign when the response plan item is executed.

3.7.1.1.7 DMSMessageImpl (Class)

The DMSMessageImpl class provides an implementation for the abstract DMSMessage class. It implements get and set methods to access and modify the MULTI-formatted message and beacon state values which make up a DMS message.

3.7.1.1.8 DMSPlanItemData (Class)

The DMSPlanItemData class is a valuetype that contains data stored in a plan item for a DMS. It is derived from PlanItemData.

3.7.1.1.9 DMSPlanItemDataImpl (Class)

The DMSPlanItemDataImpl class provides an implementation for the abstract DMSPlanItemData class. It implements get and set methods to access and modify values relative to a stored Plan Item for a DMS, which associates a stored message to a specific DMS it should be placed on.

3.7.1.1.10 DMSRPIData (Class)

The DMSRPIData class is an abstract class that describes a response plan item for a DMS. It contains the unique identifier of the DMS to contain the DMSMessage, and the DMSMessage itself.

3.7.1.1.11 DMSRPIDataImpl (Class)

The DMSRPIDataImpl class provides an implementation for the abstract DMSRPIData class. It implements get and set methods to access and modify values relative to a Response Plan Item for a DMS.

3.7.1.1.12 FP9500Configuration (Class)

The FP9500Configuration class is an abstract class that extends the CHART2DMSConfiguration class to provide configuration information specific to an

FP9500 model of DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information.

3.7.1.1.13 FP9500ConfigurationImpl (Class)

The FP9500ConfigurationImpl class provides an implementation for the abstract FP9500Configuration class. It implements get and set methods to access and modify values specific to the static configuration of an FP9500 DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information.

3.7.1.1.14 FP9500Status (Class)

The FP9500Status class is an abstract class that extends the CHART2DMSStatus class to provide status information specific to an FP9500 model of DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information. In this case, additional information provided the the FP9500 model would include things like the current message number and current message source, status bits, light status, pixel failure map, and so on.

3.7.1.1.15 FP9500StatusImpl (Class)

The FP9500StatusImpl class provides an implementation for the abstract FP9500Status class. It implements get and set methods to access and modify values specific to the dynamic status configuration of an FP9500 DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific status information.

3.7.1.1.16 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.8 HARControlModule

3.8.1 Classes

3.8.1.1 HARControlModule (Class Diagram)

This class diagram shows classes that support the use of Highway Advisory Radio (HAR) devices in the CHART II system. Details are only shown for classes that exist specifically for HAR control. Auxillary classes used from other various utility or system interface packages are shown by name only.

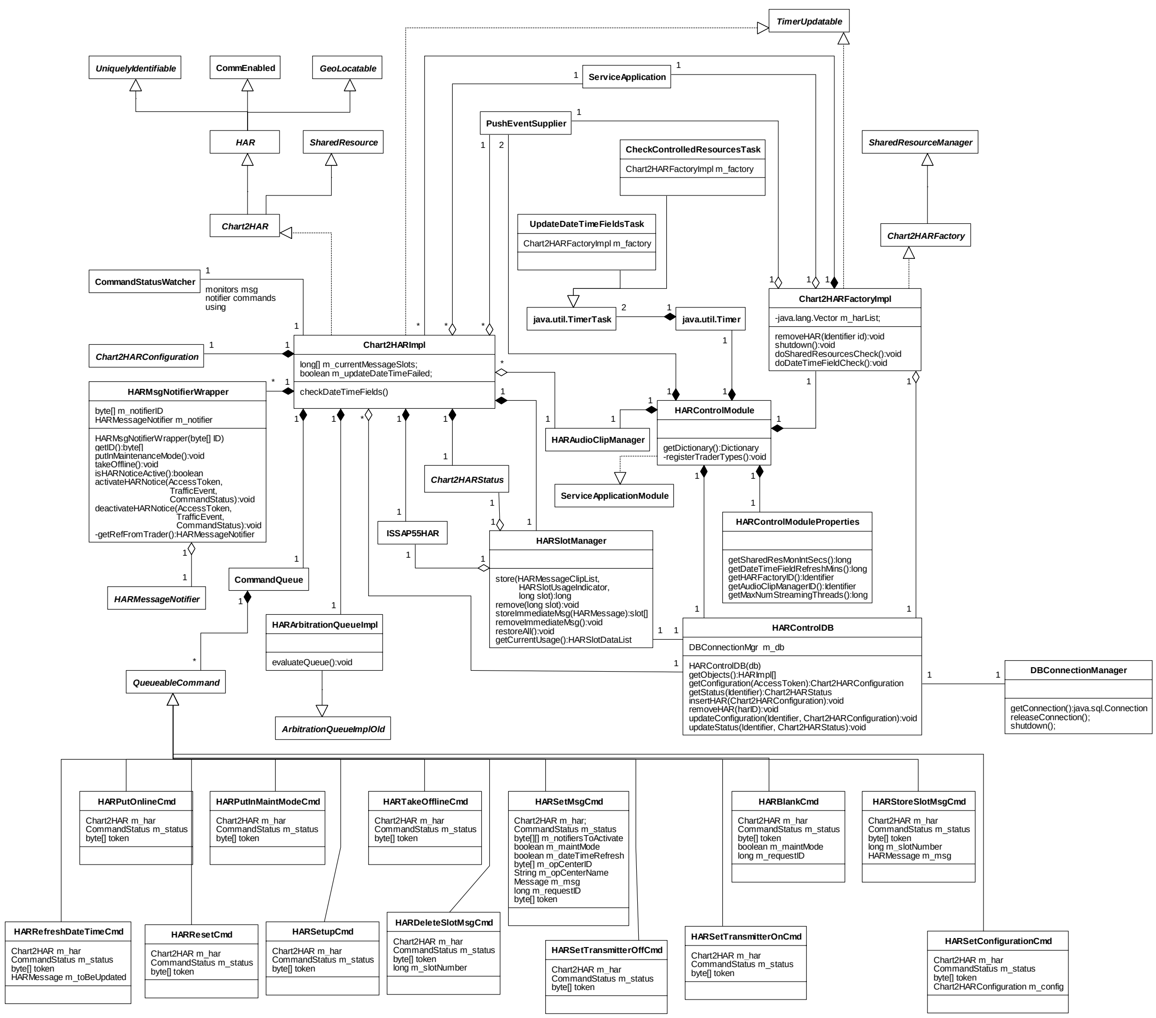

Figure 65. HARControlModule (Class Diagram)

3.8.1.1.1 CHART2HAR (Class)

The CHART2HAR class is an extension of the HAR that is aware of CHART2 business rules, such as arbitration queues, linking device usage to traffic events, and the concept of a shared resource.

3.8.1.1.2 CHART2HARConfiguration (Class)

This class contains configuration data for the HAR that is used for CHART II specific processing (as opposed to the configuration values contained in HARConfiguration that relate to typical HAR usage).

3.8.1.1.3 CHART2HARFactory (Class)

This interface defines objects capable of creating CHART2HAR objects. This factory is also responsible for monitoring the HARs as shared resources and must report when a HAR that is currently broadcasting a message (other than the default) does not have a user logged into the system that is from the controlling operations center.

3.8.1.1.4 CHART2HARFactoryImpl (Class)

This class implements the CHART2HARFactory interface as defined by the IDL specified in the System Interfaces section.

3.8.1.1.5 CHART2HARImpl (Class)

This class implements CHART2HAR as defined by IDL specified in the System Interfaces section.

3.8.1.1.6 CHART2HARStatus (Class)

This class contains status information for a CHART2HAR object. This information is specific to CHART II processing and extends beyond the status related to typical HAR device control.

3.8.1.1.7 CheckControlledResourcesTask (Class)

This class is a timer task that is executed periodically by a timer. When the run method in this class is called, it calls the CHART2HARFactoryImpl's doSharedResourcesCheck() method, which causes the factory to evaluate each HAR in the factory and determine if all HARs with a controlling op center have at least one user logged in at the op center.

3.8.1.1.8 CommandQueue (Class)

The CommandQueue class provides a queue for QueuableCommand objects. The CommandQueue has a thread that it uses to process each QueuableCommand in a first in first out order. As each command object is pulled off the queue by the CommandQueue's thread, the command object's execute method is called, at which time the command performs its intended task.

3.8.1.1.9 CommandStatusWatcher (Class)

This class is a utility that monitors one or more command status objects for completion. It periodically checks each command status object's completion code and maintains statistics on the number of failures and successes. It provides a blocking method that waits for all command status objects to complete.

3.8.1.1.10 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.8.1.1.11 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two separate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.8.1.1.12 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.8.1.1.13 HAR (Class)

This class is used to represent a Highway Advisory Radio (HAR) device. A HAR is used to broadcast traffic related information over a localized radio transmitter, making the information available to the traveler.

3.8.1.1.14 HARArbitrationQueueImpl (Class)

This class extends the ArbitrationQueueImpl to provide an implementation of its evaluateQueue() abstract method. The implementation of evaluateQueue creates a HARSetMsgCmd command and adds it to an ArbQueueMsg when a message added to the queue is to be activated on the HAR.

3.8.1.1.15 HARAudioClipManager (Class)

This class provides the implementation of the AudioStreamer interface and is capable of streaming recorded audio clips that have been previously stored. When requested to stream an audio clip, this class pulls the audio data from its persistent store pushes the audio data to the given AudioPushConsumer in a worker thread. This class also allows newly recorded audio clips to be added to the system. When a clip is added to the system it is assigned a unique ID and a HARMessageAudioClip is created as a thin wrapper to provide access to the audio data. When new audio clips are added to the system, the ID of the owner is passed to facilitate clean-up of the clip when it is no longer needed.

3.8.1.1.16 HARBlankCmd (Class)

This command object is used to blank the message on the HAR, which involves setting the message to the HAR's default message.

3.8.1.1.17 HARControlDB (Class)

This class contains all the database interaction for the HARControlModule. This class provides the ability to retrieve all HAR information on initialization, update of the configuration and status information, and insert or remove a HAR device from the system.

3.8.1.1.18 HARControlModule (Class)

This class implements the ServiceApplicationModule interface, providing a platform for publishing CHART2HAR and CHART2HARFactory objects within a service application.

3.8.1.1.19 HARControlModuleProperties (Class)

This class contains settings from a properties file used to specify parameters to be used by objects within the HARControlModule for the current instance of the application. These settings are read during the module initialization. The module must be re-started to apply any changes made to the properties file.

3.8.1.1.20 HARDeleteSlotMsgCmd (Class)

This class is used to hold data necessary to execute a request to delete a message from a slot on the HAR device.

3.8.1.1.21 HARMessageNotifier (Class)

The HARMessageNotifier class specifies an interface to be implemented by devices that can be used to notify the traveler to tune in to a radio station to hear a traffic message being broadcast by a HAR. A HARMessageNotifier is directional and allows users of the device to better determine if activation of the device is warranted for the message being broadcast by the HAR. This interface can be implemented by SHAZAMs and by DMS devices that are allowed to provide a SHAZAM-like message in the absence of any more useful messages to display.

3.8.1.1.22 HARMsgNotifierWrapper (Class)

This wrapper class is used to wrap HAR message notifiers associated with a HAR. This class handles finding the reference of the notifier object given only the object's ID. The object discovery is done at the point of first use or if a currently held reference produces a CORBA failure when used.

3.8.1.1.23 HARPutInMaintModeCmd (Class)

This class contains data needed to execute a request to put a HAR into maintenance mode.

3.8.1.1.24 HARPutOnlineCmd (Class)

This class contains data needed to execute a request to put a HAR online.

3.8.1.1.25 HARRefreshDateTimeCmd (Class)

This class contains data needed to execute a request to update the date/time fields in a message that is playing on the HAR device.

3.8.1.1.26 HARResetCmd (Class)

This class contains data needed to execute a request to reset a HAR controller.

3.8.1.1.27 HARSetConfigurationCmd (Class)

This class contains data needed to execute a request to change the configuration values of a HAR.

3.8.1.1.28 HARSetMsgCmd (Class)

This class contains data needed to execute a request to set the message played on a HAR. A flag is used to indicate if the message was set via a maintenance mode command or via the arbitration queue.

3.8.1.1.29 HARSetTransmitterOffCmd (Class)

This class contains data needed to execute a request to turn off the transmitter of a HAR device.

3.8.1.1.30 HARSetTransmitterOnCmd (Class)

This class contains data needed to execute a request to turn on the transmitter of a HAR device.

3.8.1.1.31 HARSetupCmd (Class)

This class contains data needed to execute a request to issue the setup command for the HAR.

3.8.1.1.32 HARSlotManager (Class)

This class manages the slot usage for the CHART2HARImpl. When a clip is to be stored in the HAR controller, this class is called instead of calling the ISSAP55HAR directly. This class ensures the reserved slot numbers (default header, default trailer, default message, current message) are not overlaid with other clips stored in the controller. When clips are stored in slots in the controller, this class keeps track of the run-time for each and the total run time for the device and provides an error when the storage of a clip exceeds the configured available run time of the device.

This class also helps to manage the condition when multiple slots are needed for the current (immediate) message. This will be true if the current message consists of 3 or more clips and a pre-stored clip exists and is preceded and followed by a text or voice clip.

3.8.1.1.33 HARStoreSlotMsgCmd (Class)

This class contains data needed to execute a request to store a message clip into a slot within the HAR controller.

3.8.1.1.34 HARTakeOfflineCmd (Class)

This class contains data needed to execute a request to take a HAR offline.

3.8.1.1.35 ISSAP55HAR (Class)

This class contains the model specific implementation of HAR features supported by the Information System Specialists (ISS) AP55 HAR controller. This class stores no data related to the current state of the device. Instead, this class is used to encapsulate the device protocol and acts as a utility class to enable an application level class to control the AP55 without communications knowledge.

This class uses the TelephonyManager to acquire a port when needed. This class must handle cases when a telephony manager has all ports busy. It could wait for a port to become available or seek out another telephony manager in the system and attempt to aquire one of its ports.

3.8.1.1.36 java.util.Timer (Class)

This class provides asynchronous execution of tasks that are scheduled for one-time or recurring execution.

3.8.1.1.37 java.util.TimerTask (Class)

This class is an abstract base class which can be scheduled with a timer to be executed one or more times.

3.8.1.1.38 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier’s push rate.

3.8.1.1.39 QueueableCommand (Class)

A QueueableCommand is an interface used to represent a command that can be placed on a CommandQueue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed. This interface must be implemented by any device command in order that it may be queued on a CommandQueue. The CommandQueue driver calls the execute method to execute a command in the queue and a call to the interuppted method is made when a CommandQueue is shut down.

3.8.1.1.40 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.8.1.1.41 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.8.1.1.42 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.8.1.1.43 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn't have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.8.1.1.44 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.8.1.1.45 UpdateDateTimeFieldsTask (Class)

This class is a timer task that is executed periodically by a timer. When executed, the run method of this class calls the CHART2HARFactoryImpl's doDateTimeFieldCheck(), which in turn calls each HAR in the factory to have it determine if it needs to update any field messages that use date time fields.

3.8.2 Sequence Diagrams

3.8.2.1 HARControlModule:activateMessageNotifiers (Sequence Diagram)

This diagram shows the processing involved when the HAR needs to activate one or more of its message notifiers. Because message notifiers process asynchronously, each message notifier is told to activate or deactivate and a CommandStatusWatcher is used to track the progress of the notifiers.

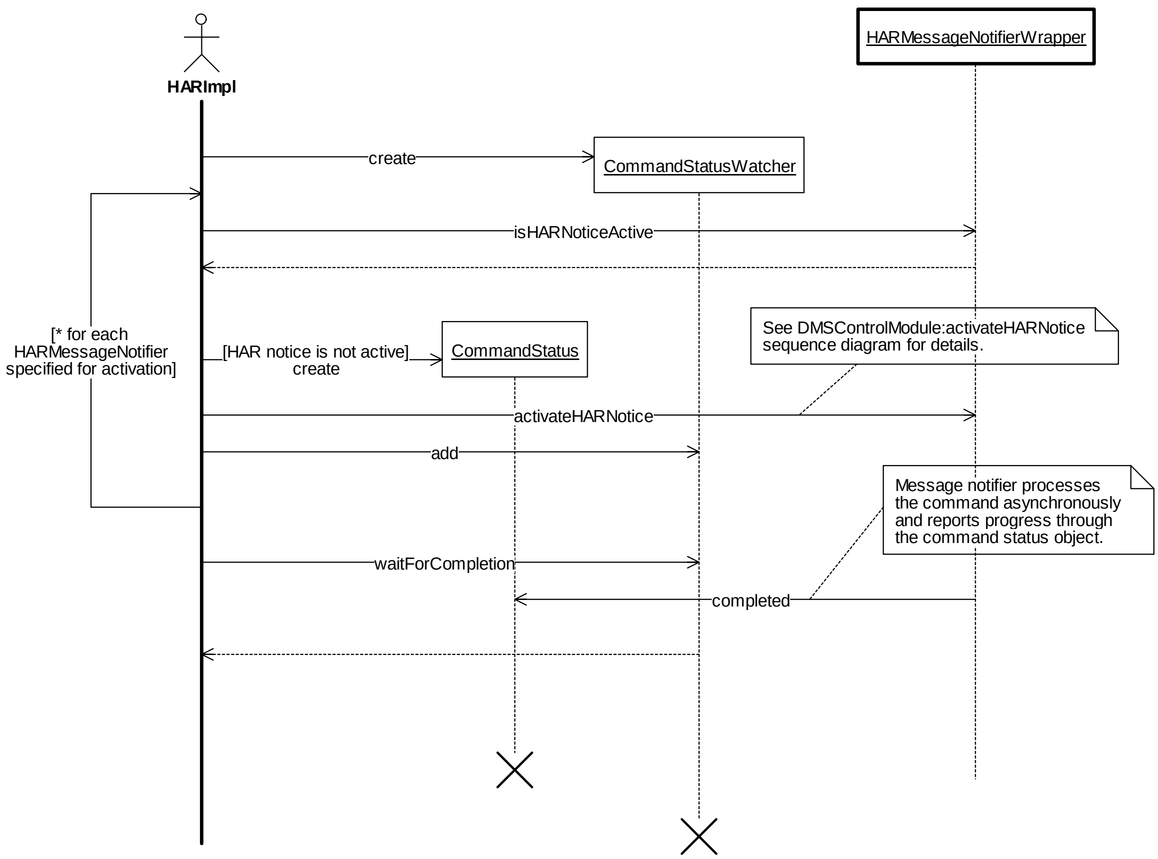

Figure 66. HARControlModule:activateMessageNotifiers (Sequence Diagram)

3.8.2.2 HARControlModule:addEntry (Sequence Diagram)

The addEntry method defined in the ArbitrationQueue interface is used to put a message on a HAR when the HAR is online. The ArbQueueProcessing:addEntry sequence diagram shows the processing that occurs that is generic in nature, for the arbitration queue base class implementation is shared and is not HAR specific. Part of the base class processing of addEntry involves calling the evaulateQueue method. Because this method is abstract, the derived class provides the implementation of this method. In the case of a HAR, the derived class is a HARArbitrationQueueImpl. The details of the HARArbitrationQueueImpl's evaluateQueue processing are shown in the HARControlModule:evaluateQueue sequence diagram.

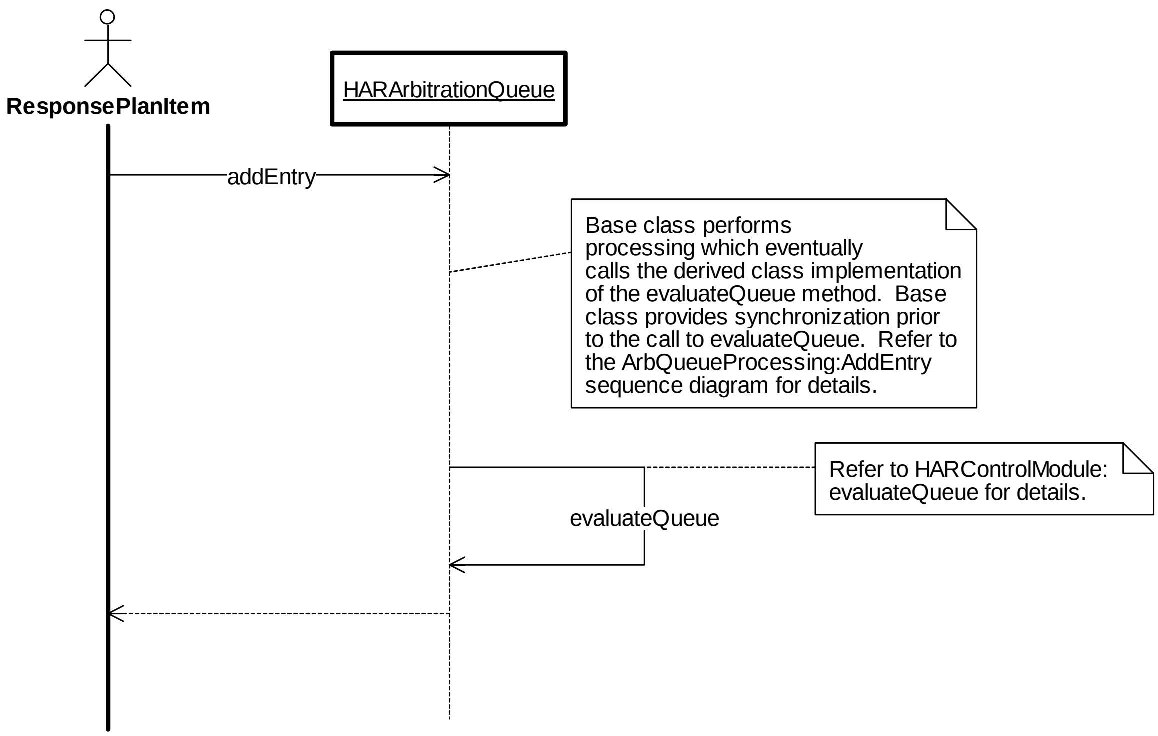

Figure 67. HARControlModule:addEntry (Sequence Diagram)

3.8.2.3 HARControlModule:blank (Sequence Diagram)

A user with proper functional rights can blank a HAR when it is in maintenance mode. This command is executed asynchronously by placing a HARBlankCmd on the CommandQueue. When the command queue executes this command, the blankImpl method is invoked on the HAR. Refer to the HARControlModule:blankImpl sequence diagram for details.

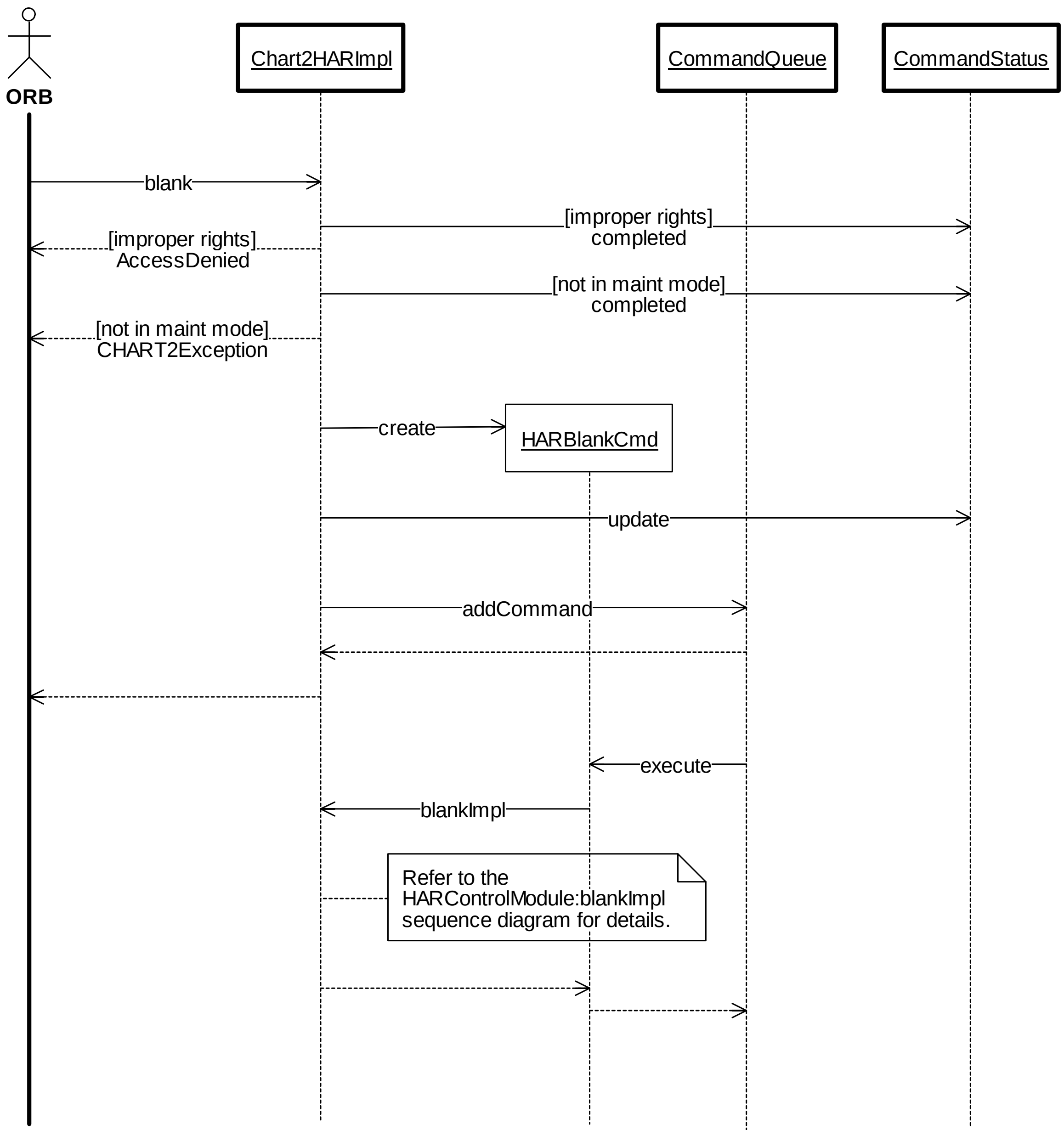

Figure 68. HARControlModule:blank (Sequence Diagram)

3.8.2.4 HARControlModule:blankImpl (Sequence Diagram)

The sequence diagram shows the processing that occurs when a HARBlankCmd is executed. This command is placed on the command queue by the HAR blank method when in maintenance mode or by the arbitration queue's removeEntry method. A flag in the command object is used to distinguish the origin of the command to allow for the proper mode check to be done and to allow for specific processing that is to be done when the HAR is blanked by the arbitration queue.

The HAR is blanked using the ISSAP55HAR object and having it command the HAR to play the message in its default message slot. If the default message is successfully set to be played, if any previous immediate message existed it is removed from the HAR slot(s) it occupied and any recorded voice data used in the previous immediate message is removed from the system.

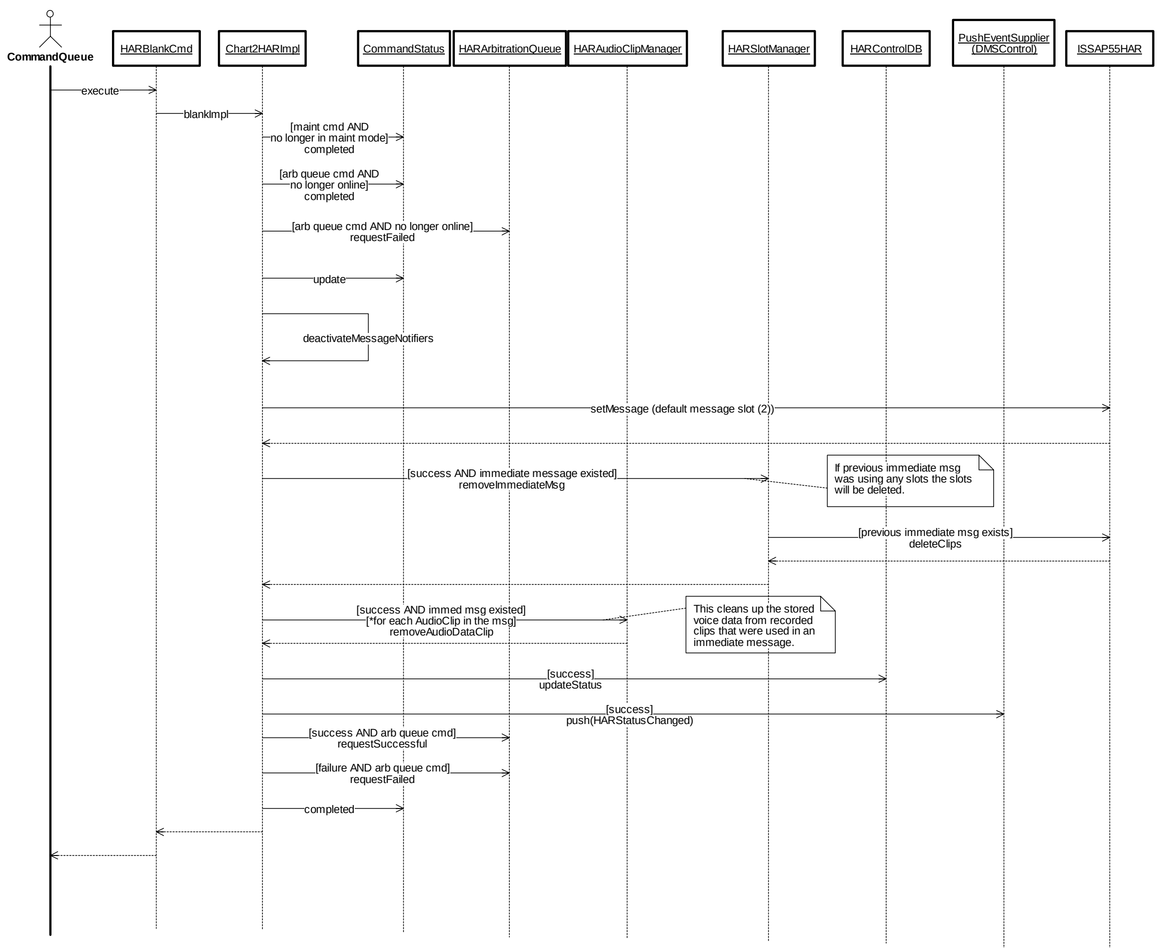

Figure 69. HARControlModule:blankImpl (Sequence Diagram)

3.8.2.5 HARControlModule:Shutdown (Sequence Diagram)

When the HARControlModule is shut down by the ServiceApplication, it stops its timer based processing, disconnects its objects from the ORB, and releases any resources it is using.

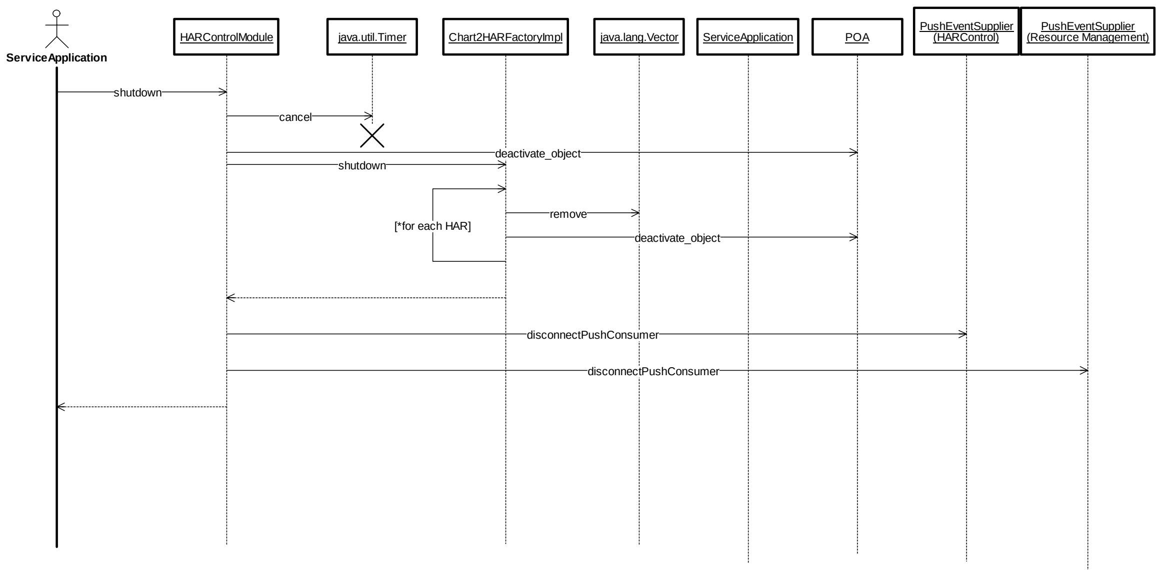

Figure 70. HARControlModule:Shutdown (Sequence Diagram)

3.8.2.6 HARControlModule:createHAR (Sequence Diagram)

A user with the proper functional rights can add a HAR to the system. The HAR object is created by the HARControlDB object, which takes care of adding the appropriate data to the database and constructing a CHART2HARImpl object. The factory connects the object to the ORB, registers it with the ServiceApplication (which causes the object to be published in the trader), and pushes an event to notify others that a HAR has been added to the system. The HAR is added in offline mode and therefore no field communications are necessary.

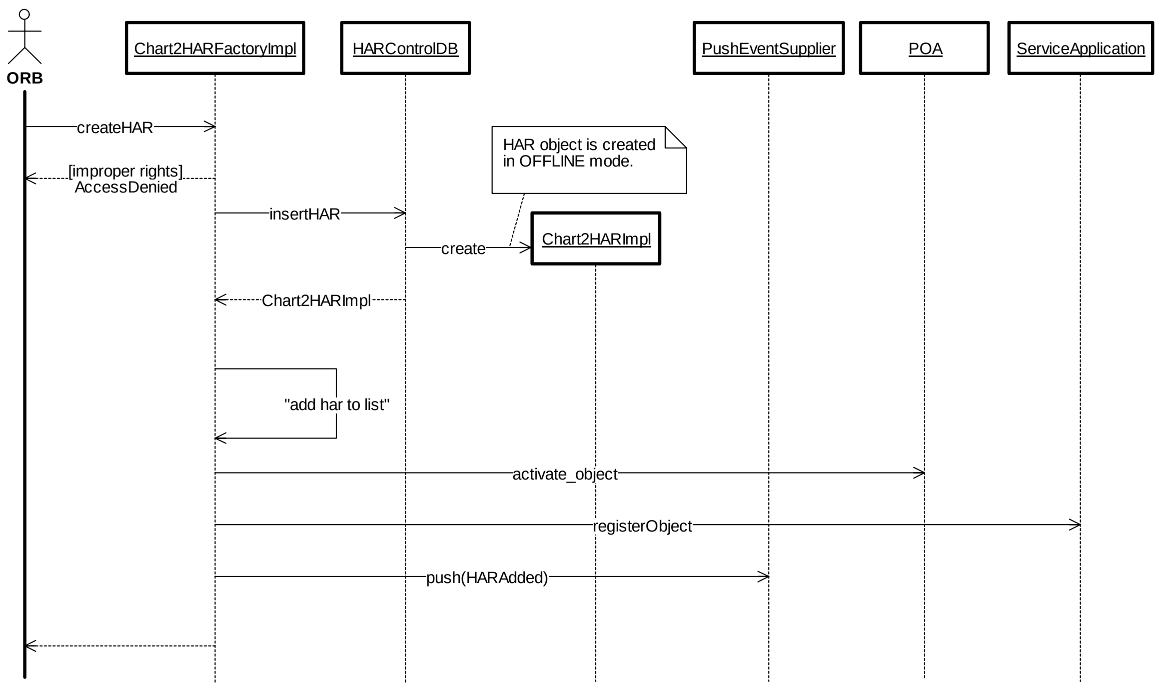

Figure 71. HARControlModule:createHAR (Sequence Diagram)

3.8.2.7 HARControlModule:deactivateMessageNotifiers (Sequence Diagram)

This diagram shows the processing that occurs when the HAR deactivates its associated message notifiers. Because the message notifiers process their deactivate command asynchronously, the CHART2HARImpl uses a CommandStatusWatcher to monitor the command status objects passed to each notifier and determine the status of the operation.

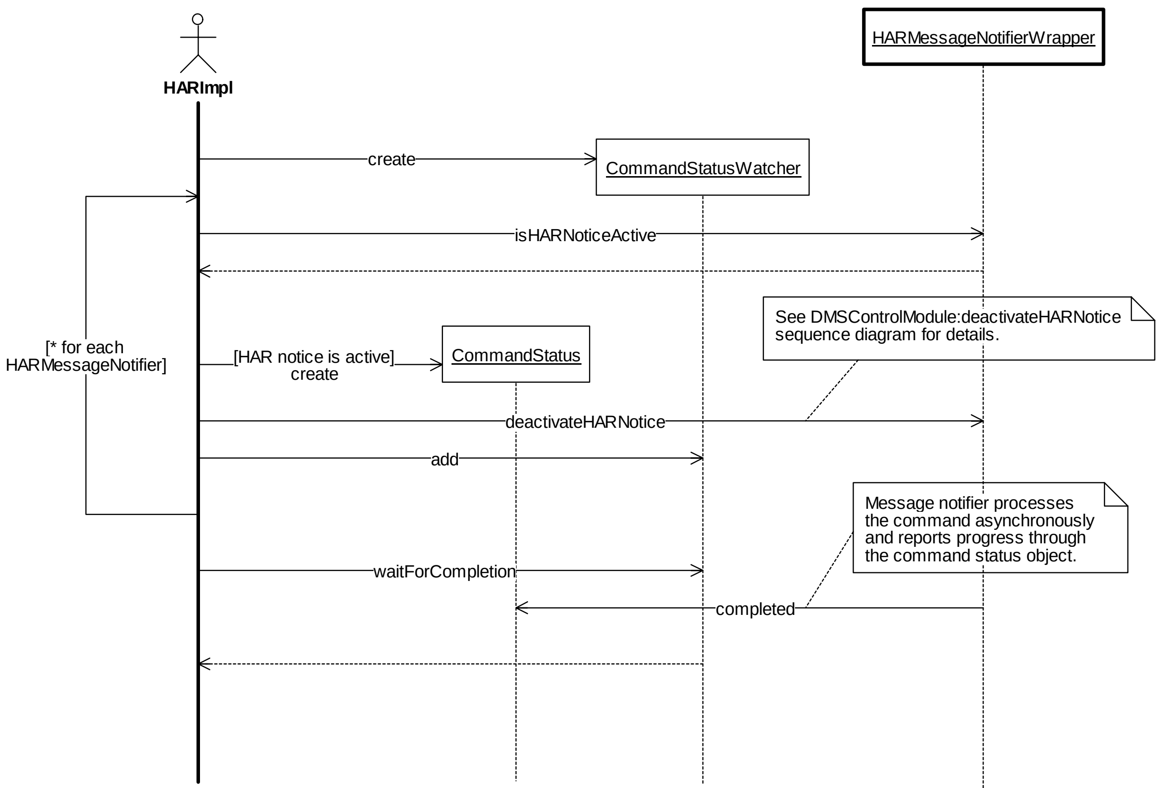

Figure 72. HARControlModule:deactivateMessageNotifiers (Sequence Diagram)

3.8.2.8 HARControlModule:deleteSlotMessage (Sequence Diagram)

This diagram shows the processing involved when a message that was previously stored in a slot on the HAR controller is deleted. The command is processed asynchronously via the command queue. In addition to deleting the message from the slot on the HAR controller, any voice data that was custom recorded for in the message may be removed from the system. The voice data is not removed from the system if a message library originally stored it.

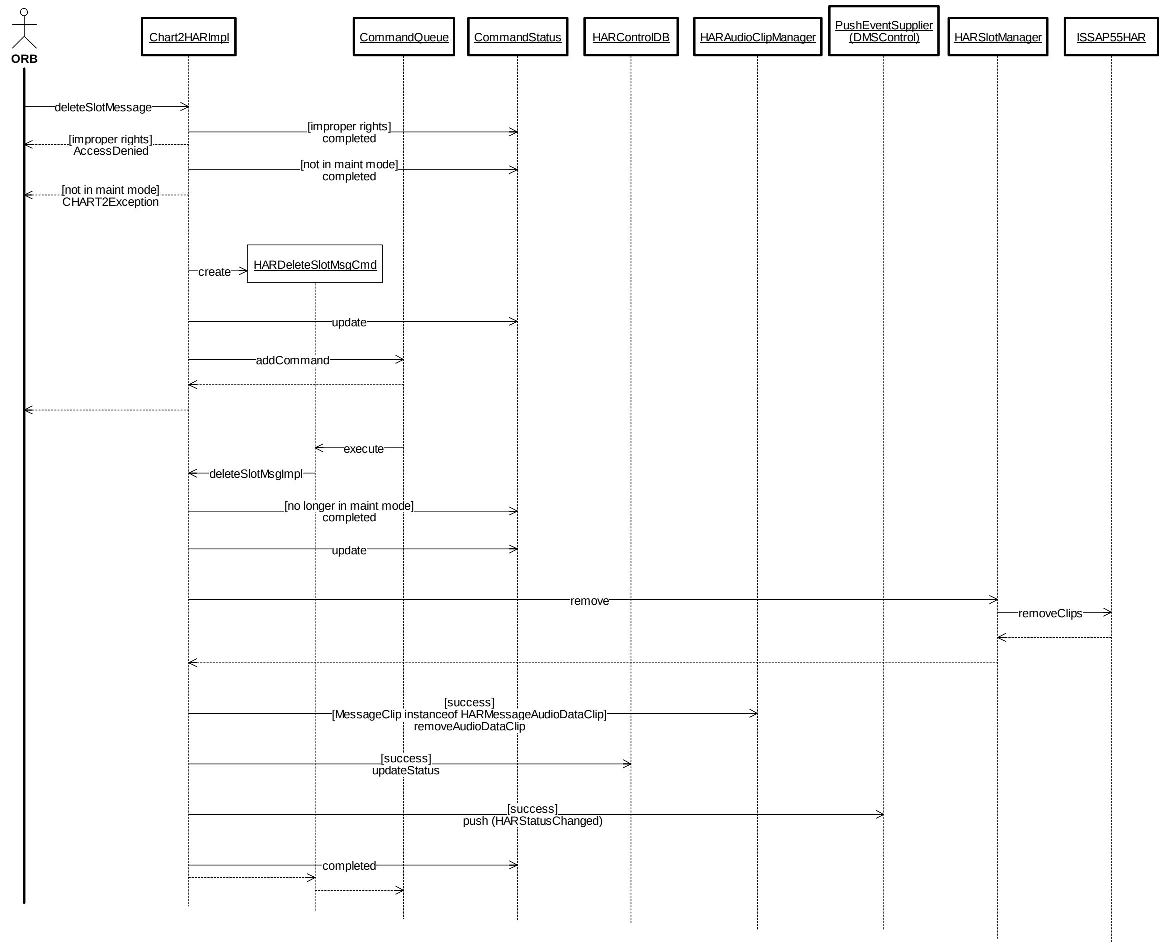

Figure 73. HARControlModule:deleteSlotMessage (Sequence Diagram)

3.8.2.9 HARControlModule:evaluateQueue (Sequence Diagram)

This diagram shows the processing done by the HARArbitrationQueueImpl's implementation of the ArbitrationQueueImpl's evaluateQueue abstract method. The base class implementation performs housekeeping prior to calling evaluateQueue, so the evaluate queue only needs to evaluate the messages on the message queue and determine the message (or messages) to put on the device or determine if the device should be blanked. In this implementation, at most one message is on the arbitration queue for activation. When told to evaluate the queue, the HARArbitrationQueue looks at the top entry on the queue to decide the processing that must occur. If the entry is not already active or is marked for update and it is not marked for deletion, the message contained in the entry will be set on the HAR. If the entry is marked for deletion and is active, the HAR is blanked. Refer to the HARControlModule:setMessageImpl and HARControlModule:blankImpl sequence diagrams for details on the processing that occurs when the arbitration queue executes the command.

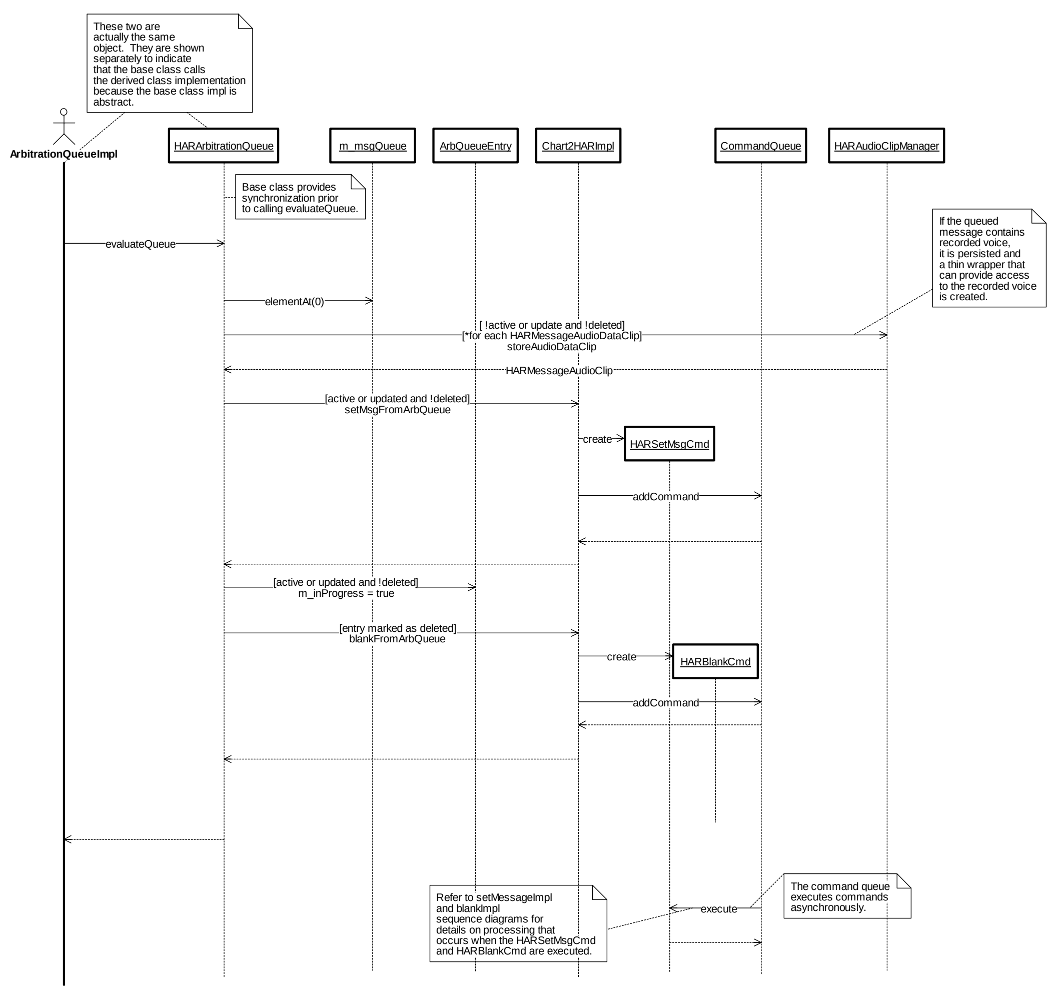

Figure 74. HARControlModule:evaluateQueue (Sequence Diagram)

3.8.2.10 HARControlModule:getConfiguration (Sequence Diagram)

A user with appropriate priveleges can get the current configuration of the HAR. This involves returning the current configuration object from the HAR object.

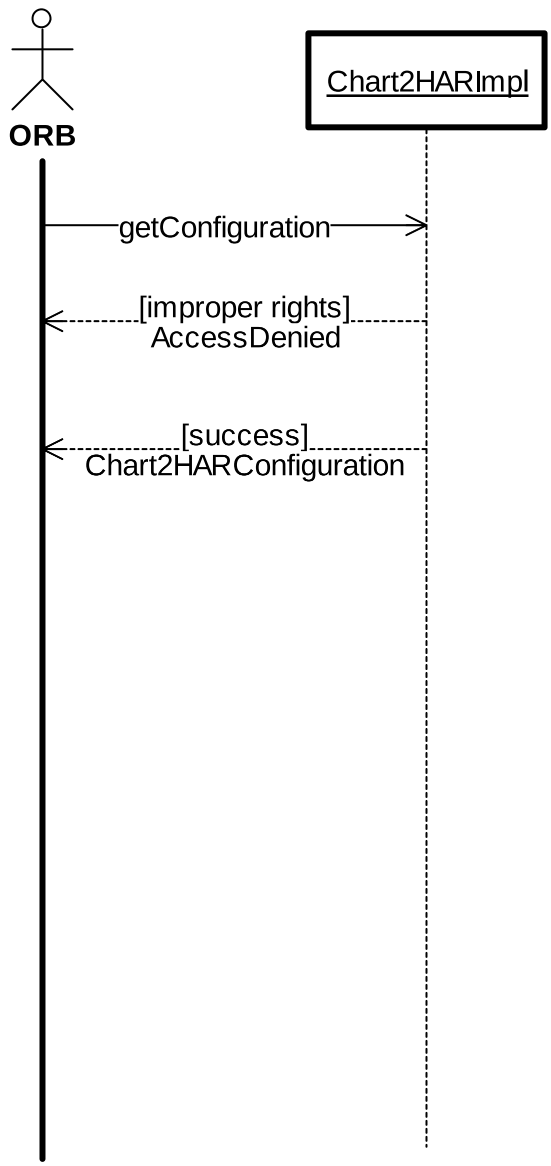

Figure 75. HARControlModule:getConfiguration (Sequence Diagram)

3.8.2.11 HARControlModule:getStatus (Sequence Diagram)

When a request is made for the current status of the HAR, the HAR's status object is returned.

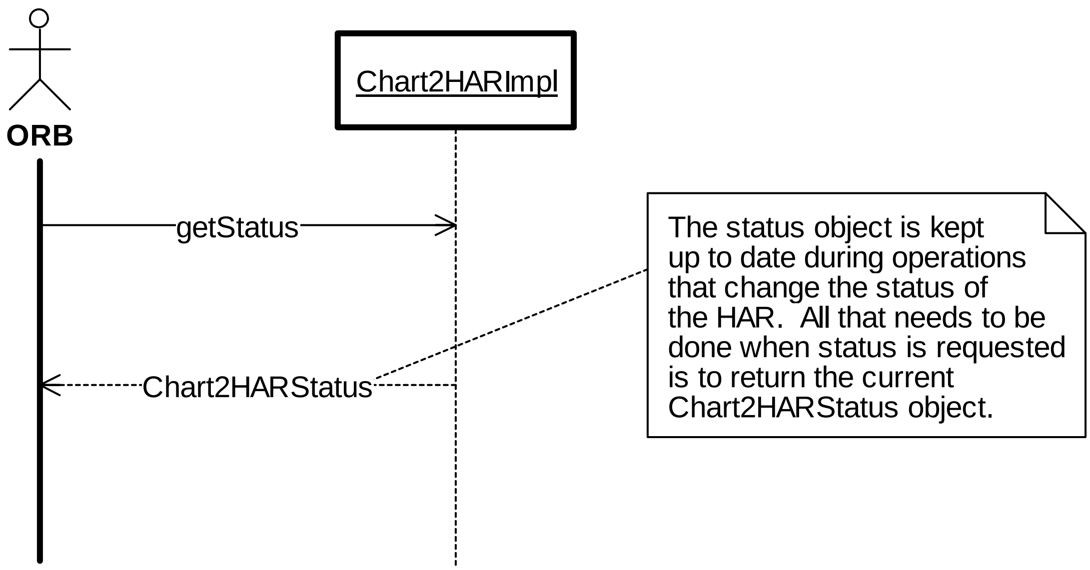

Figure 76. HARControlModule:getStatus (Sequence Diagram)

3.8.2.12 HARControlModule:Initialize (Sequence Diagram)

This sequence diagram shows the processing that takes place when the HARControlModule is initialized. The module creates the support objects that will be needed by the HAR factory and the HAR objects. The HAR Factory is created which in turn creates the HARs that have been previously added to the factory. The factory and the HAR objects are added to a recurring timer so that they can conduct their timer based processing when appropriate. The factory performs shared resource management checks periodically and the HARs may need to periodically update their message based on the time of day, depending on the message content.

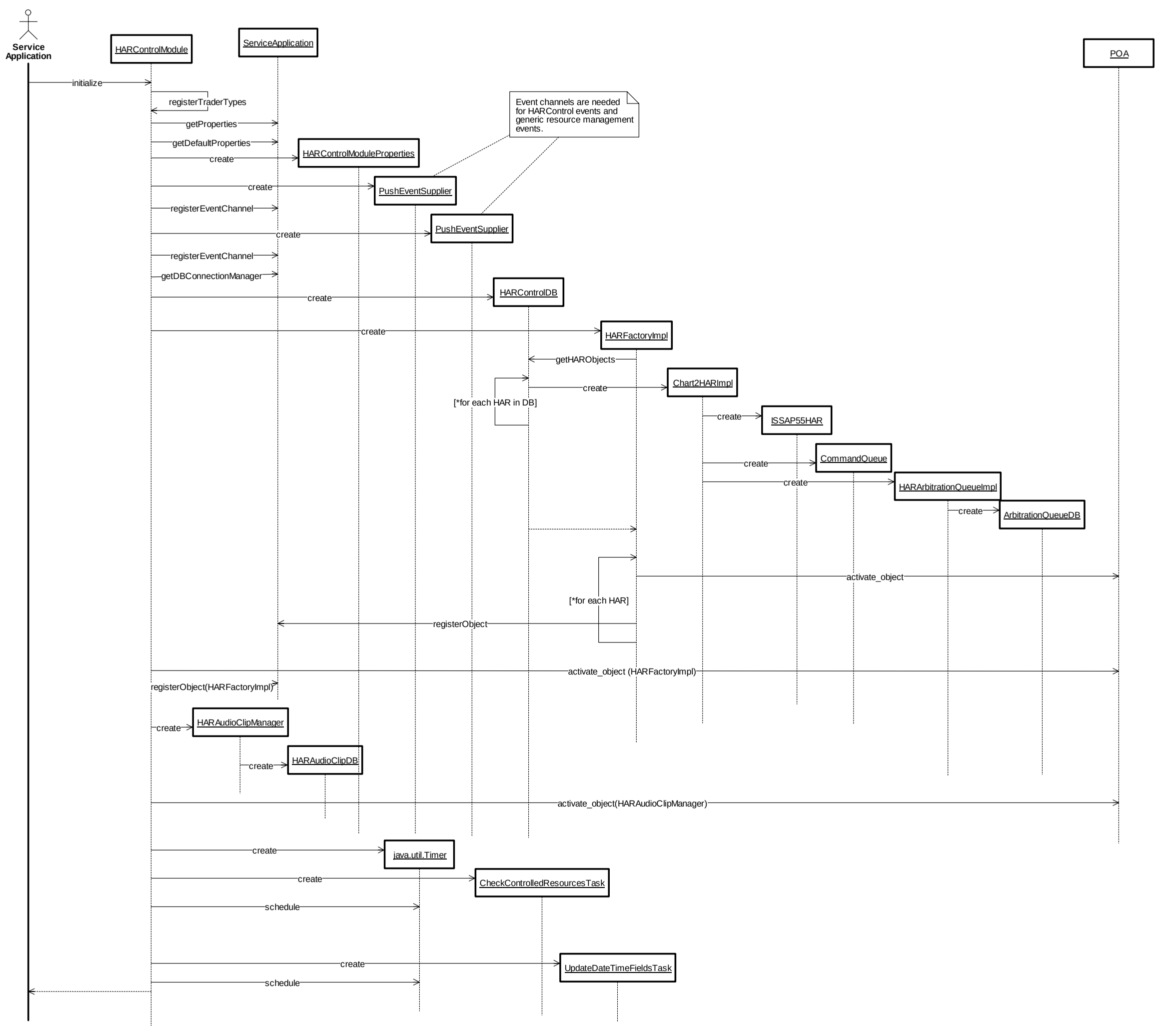

Figure 77. HARControlModule:Initialize (Sequence Diagram)

3.8.2.13 HARControlModule:PutInMaintenanceMode (Sequence Diagram)

A user with appropriate priveleges can put a HAR in maintenance mode. When this occurs, the HAR is blanked and its transmitter is turned off. If there is a failure commanding the device, the status of the HAR is still marked as blank in CHART II and the device is moved to the maintenance mode state.

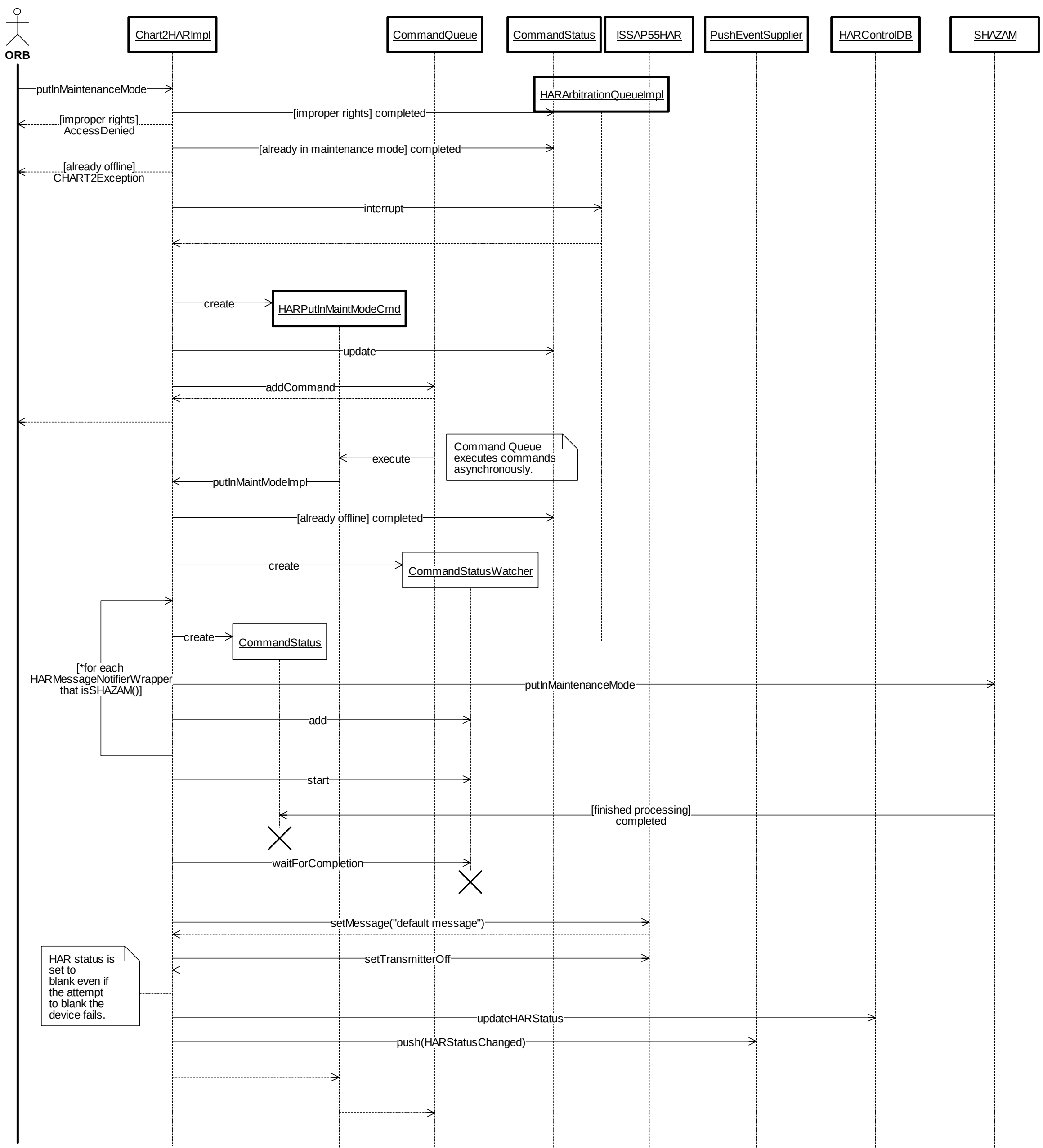

Figure 78. HARControlModule:PutInMaintenanceMode (Sequence Diagram)

3.8.2.14 HARControlModule:PutOnline (Sequence Diagram)

A user with appropriate priveleges can put a HAR online. When this occurs, the HAR is blanked and the transmitter is set on. If a failure occurs while commanding the device, the device is not brought online.

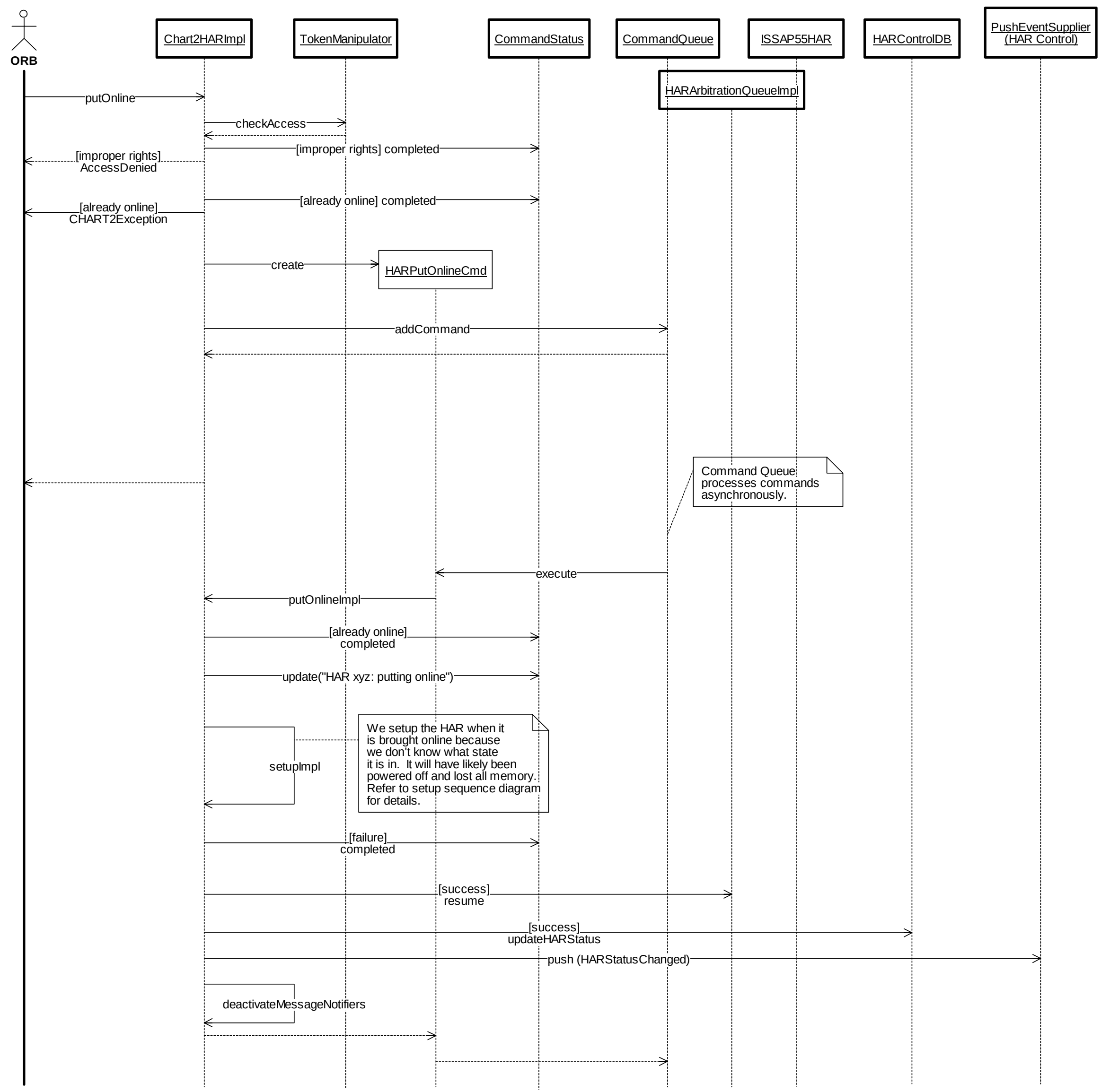

Figure 79. HARControlModule:PutOnline (Sequence Diagram)

3.8.2.15 HARControlModule:removeEntry (Sequence Diagram)

Remove entry is called when a message placed on an arbitration queue is no longer needed by the originating traffic event. The base class performs queue housekeeping and then calls the derived class's implemenation of evaluateQueue. Refer to the ArbQueueProcessing:removeEntry sequence diagram for details. The processing performed by the HARArbitrationQueueImpl's evaluateQueue method is shown in HARControlModule:evaulateQueue.

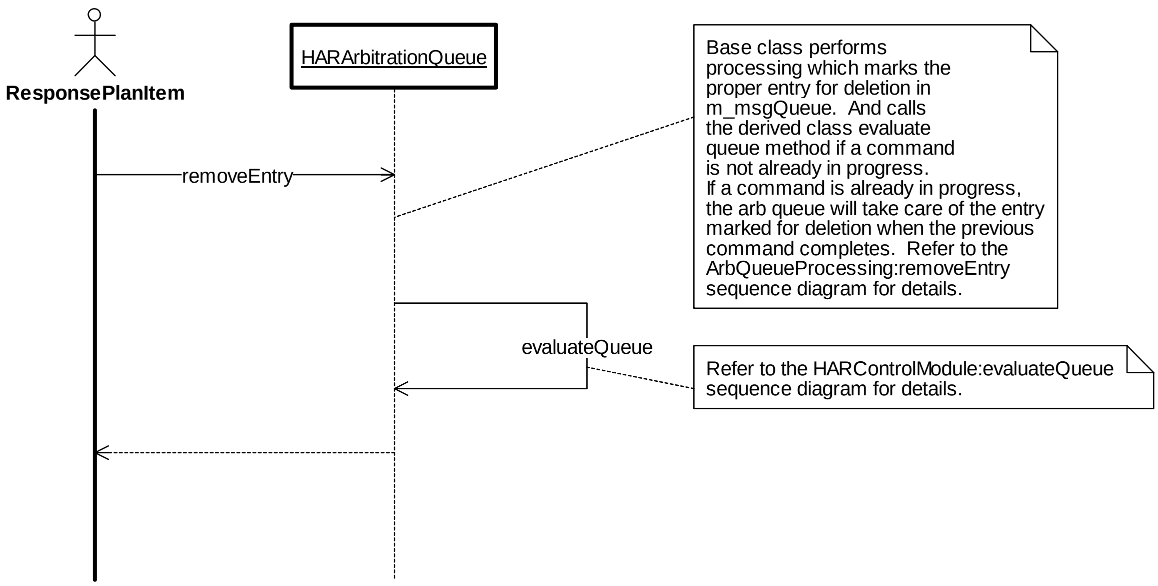

Figure 80. HARControlModule:removeEntry (Sequence Diagram)

3.8.2.16 HARControlModule:removeHAR (Sequence Diagram)

A user with proper functional rights can remove a HAR from the system if the HAR is offline. The HAR delegates its removal to the HAR factory that created it. The HAR is withdrawn from the trader and disconnected from the ORB. The HARControlDB object is called to remove the HAR from the database, and the HAR is removed from the HAR factory's list of HARs. After the HAR has been removed from the HAR list, no references to the HAR exist in the HARControlModule and the CHART2HARImpl object is deleted.

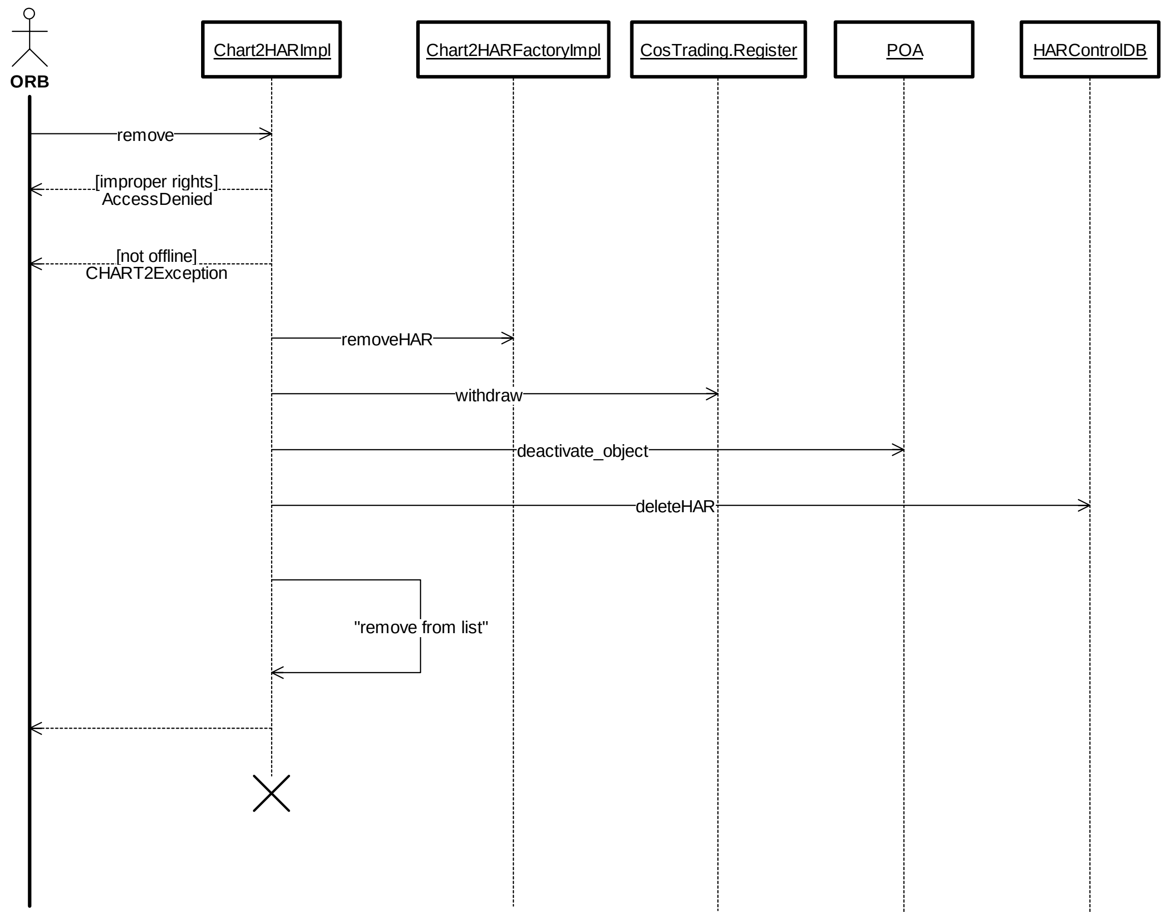

Figure 81. HARControlModule:removeHAR (Sequence Diagram)

3.8.2.17 HARControlModule:reset (Sequence Diagram)

A user with the proper functional rights can reset the HAR controller when the HAR is in maintenance mode. A reset command is issued to the HAR controller that erases all stored data in the HAR. The setupImpl method is then called to restore the data that resides on the HAR. Refer to the setup sequence diagram for details on the setupImpl call.

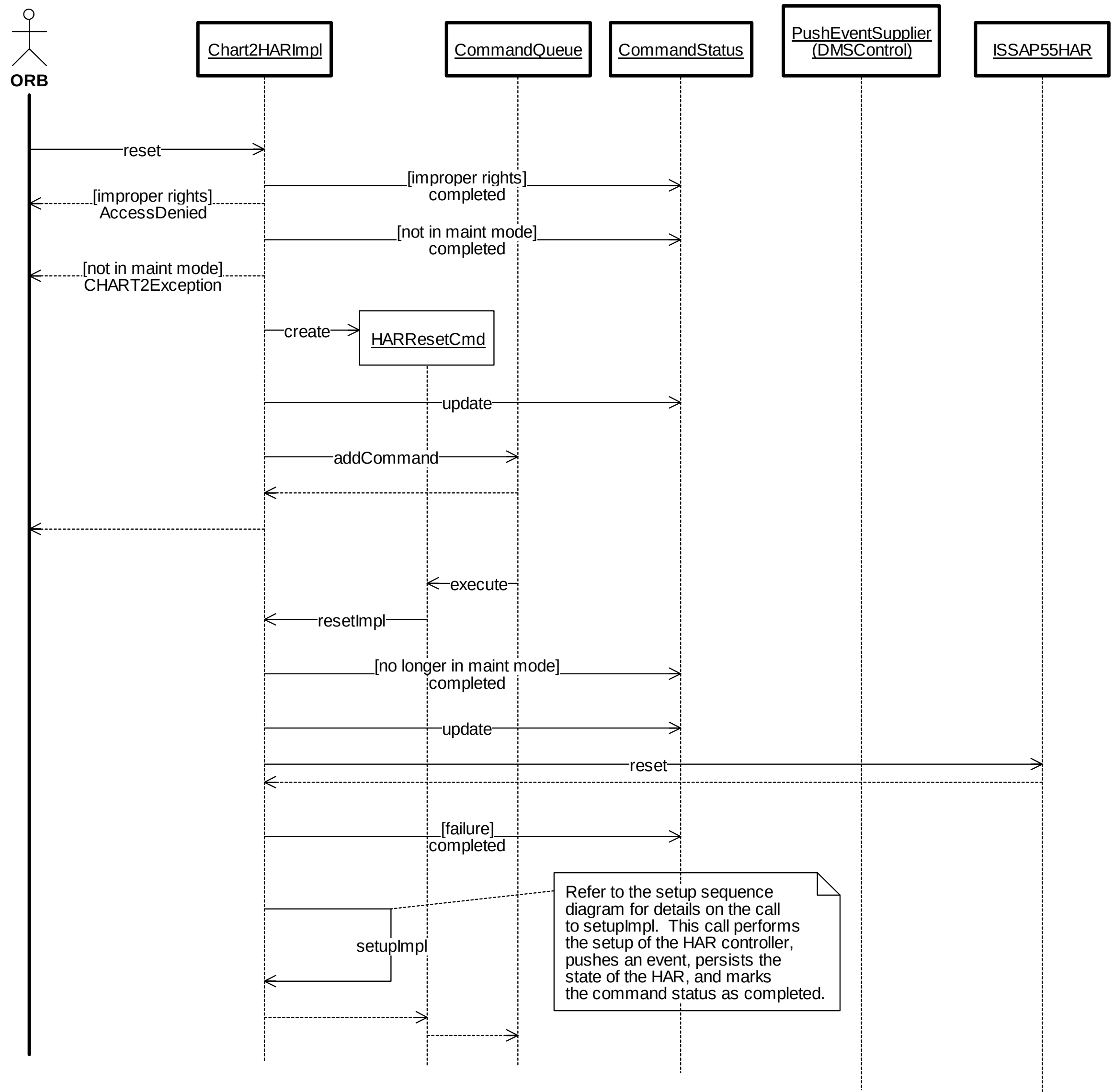

Figure 82. HARControlModule:reset (Sequence Diagram)

3.8.2.18 HARControlModule:setConfiguration (Sequence Diagram)

A user with the appropriate priveleges can set the configuration of the HAR. The HAR must be in maintenance mode when setting the configuration. The command is processed asynchronously by the command queue. Because the configuration consists of many separate values that are set individually on the device, the possiblity of partial success exists. When this occurs warning messages are given back to the user through the command status object and the configuration is set to reflect the partial success.

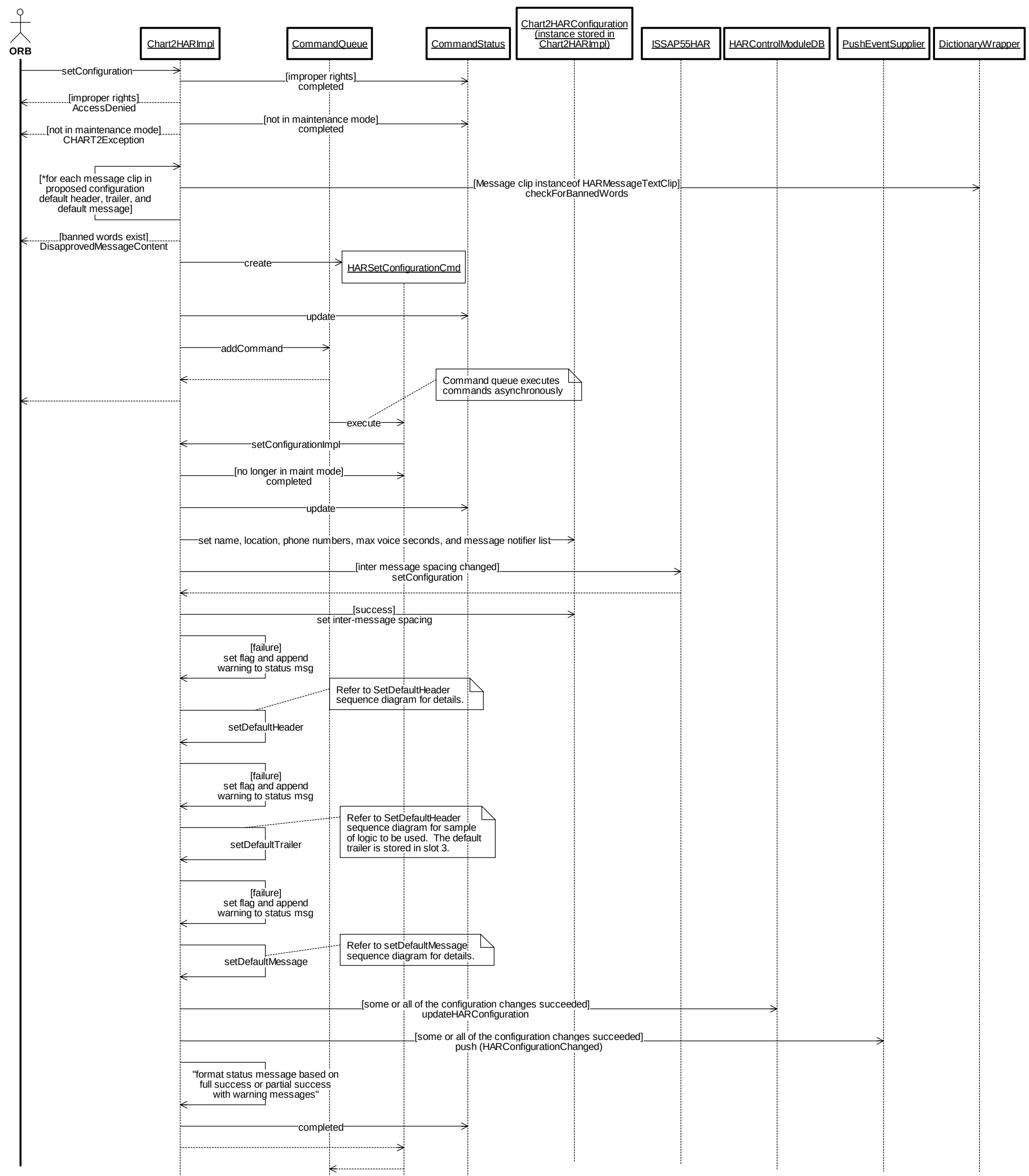

Figure 83. HARControlModule:setConfiguration (Sequence Diagram)

3.8.2.19 HARControlModule:SetDefaultHeader (Sequence Diagram)

This sequence diagram shows processing that occurs when a message clip is received by the CHART2HARImpl to be stored as the default header for the HAR. This is a sub-process of setting the configuration of the HAR. The logic for this operation also applies to setting the default trailer for a HAR.

The message clip is processed differently based on the type of message clip it is. If the message clip is a text message clip and the clip is different than the current default header for the HAR, the ISSAP55HAR object is called to store the text message (converted to speech) in slot 1 of the HAR. If the storage into the HAR is successful, the message clip is stored in the current HAR configuration as the default message header.

If the message clip is an audio data message clip (custom recorded by the operator) the voice data is persisted and a thin wrapper is created to represent the clip. This thin wrapper does not carry the voice data (which could be very large) but instead carries a reference to an object (streamer) that can supply the data when it is needed.

If the message clip passed is a pre-stored clip (used to indicate a slot in the HAR that has been previously downloaded with a message), the processing of the default header produces an error. Note that the GUI prevents the user from using a prestored clip for the default header because the default header is itself stored in a slot on the HAR, thus this check exists as an extra precaution.

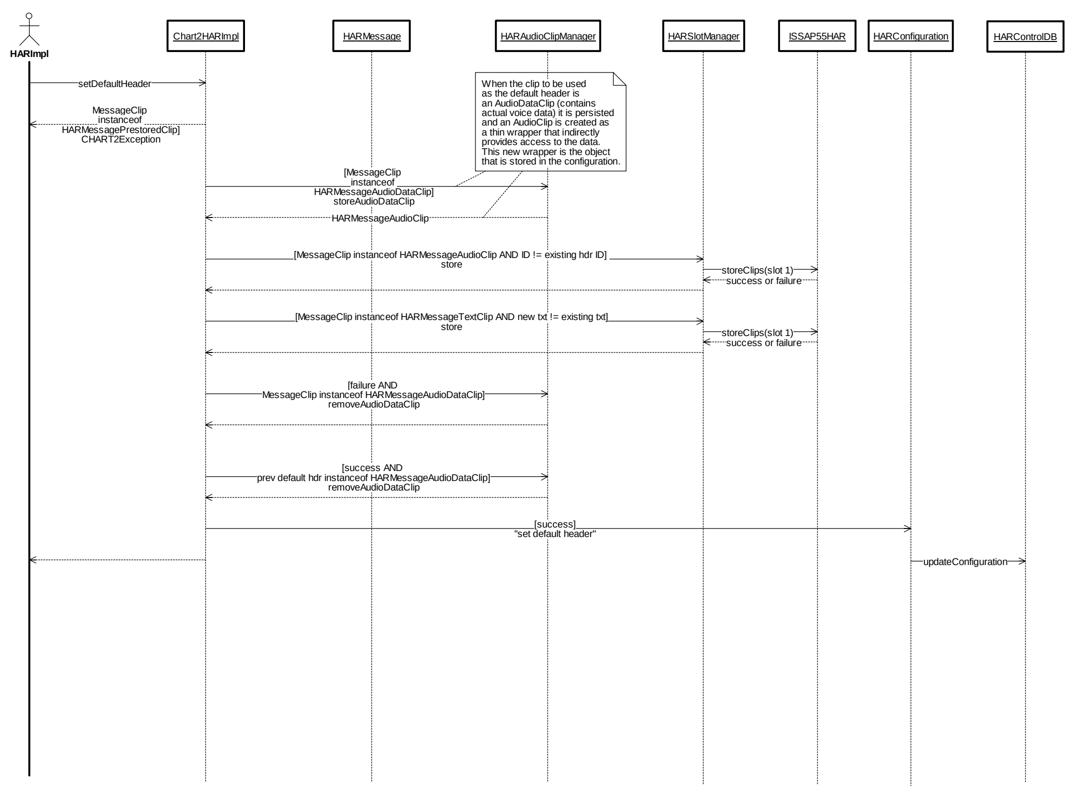

Figure 84. HARControlModule:SetDefaultHeader (Sequence Diagram)

3.8.2.20 HARControlModule:setDefaultMessage (Sequence Diagram)

This sequence diagram shows processing that occurs when a message is received by the CHART2HARImpl to be stored as the default message for the HAR. This is a sub-process of setting the configuration of the HAR. The logic for this operation also applies to setting the default trailer for a HAR.

Each clip in the header, body, and trailer of the message are processed to persist any recorded voice prior to downloading the message to the HAR. The message is then sent to the HAR via the ISSAP55 object and the HAR's configuration is updated with the new default message. Note that when storing the HAR message in the controller, if use of default header and / or trailer is specified in the message they will not be downloaded to the controller but instead default header and/or trailer slots will be specified when the default message is to be played.

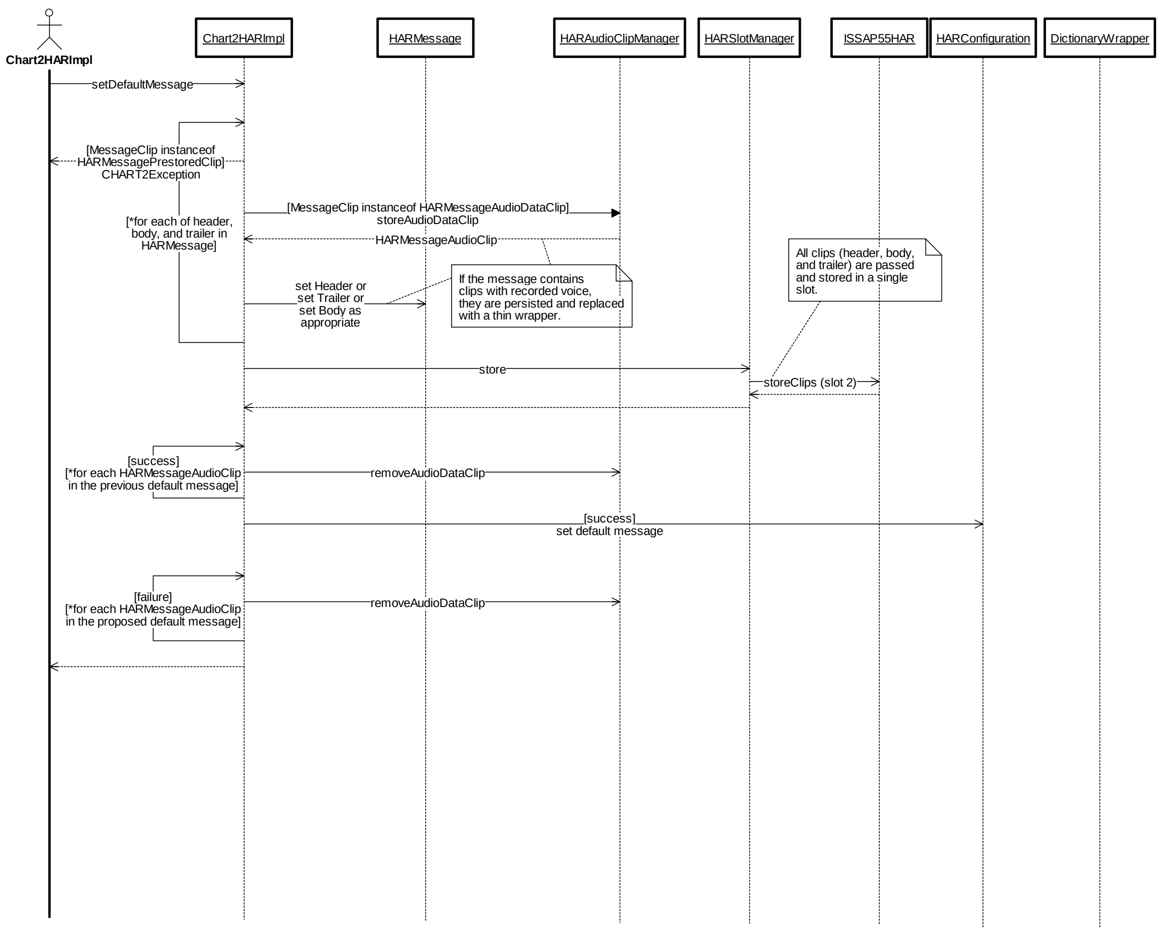

Figure 85. HARControlModule:setDefaultMessage (Sequence Diagram)

3.8.2.21 HARControlModule:setMessage (Sequence Diagram)

A user with proper functional rights can set a message on a HAR when it is in maintenance mode. A command object that knows how to set a message on a HAR is created and passed to the CommandQueue to be processed asynchronously. The processing done when the command is executed from the command queue is shared among the set message in maintenance mode and the setting of a message through a traffic event and is therefore shown on a separate diagram, HARControlModule:setMessageImpl.

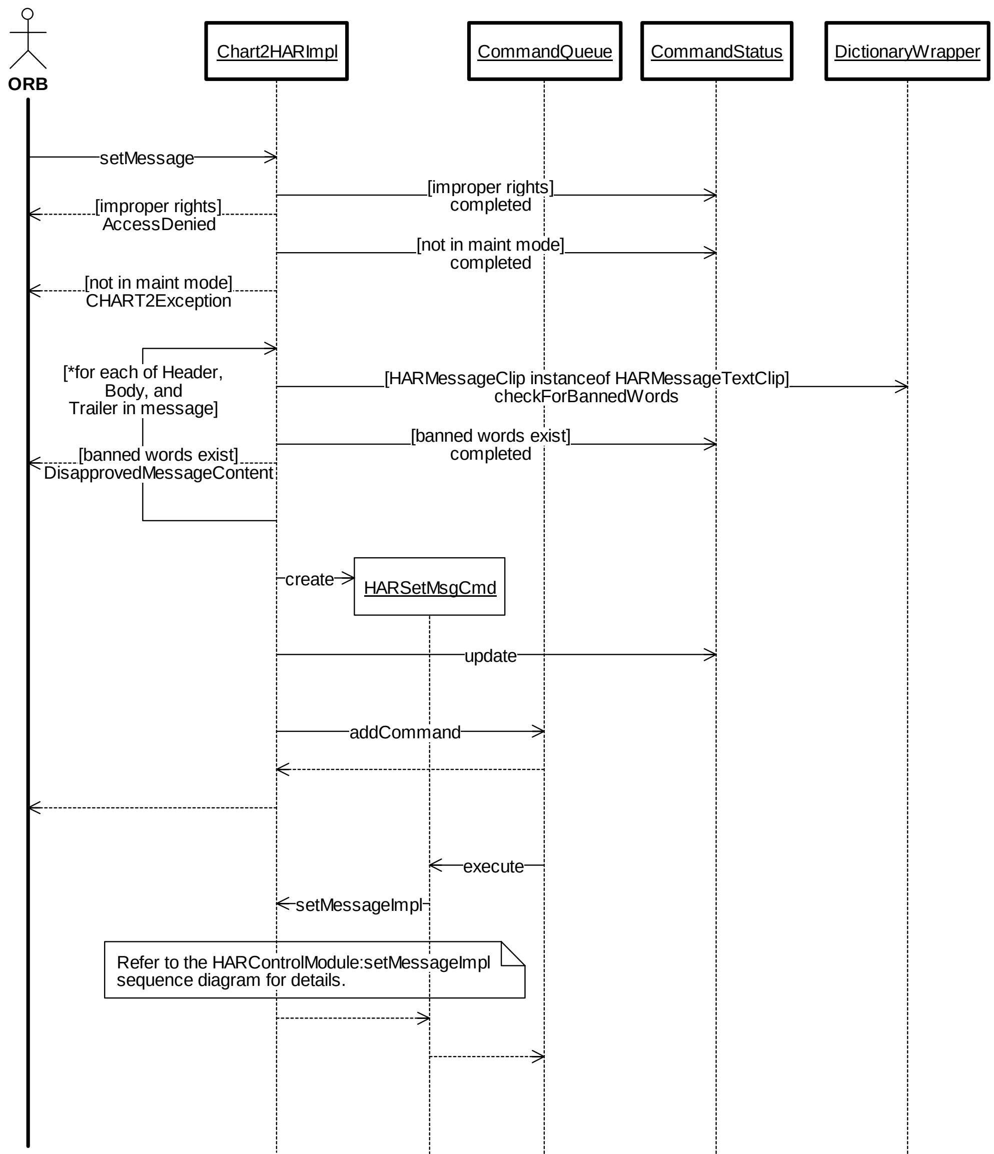

Figure 86. HARControlModule:setMessage (Sequence Diagram)

3.8.2.22 HARControlModule:setMessageImpl (Sequence Diagram)

This sequence diagram shows the processing that occurs when a HARSetMsgCmd is executed from the command queue. This command can be placed on the queue as a maintenance command or as part of online processing, therefore some of the processing differs based on origination of the message. Refer to the notes on the diagram for details.

When setting the message on the HAR, any recorded voice clips that exist in the message are passed to the HARAudioClipManager for storage and they are converted from the heavy weight HARMessageAudioDataClip objects (which contain the actual voice data) to lightweight HARMessageAudioClip objects, which contain a streamer that can provide the data when needed. These lightweight objects are used to pass voice clips throughout the system to avoid the bandwidth needed to pass the actual voice data. The actual voice data is only passed (via the streamer) when the actual voice data is needed for listening (by the end user) or for playing to the device (by FMS). Messages that are set when the device is online through the arbitration queue's addEntry method will store off any voice data in the HARAudioClipManager prior to the setMessageImpl getting invoked, so the processing done on the HARAudioClipManager shown on the diagram will only ever apply to messages set in maintenance mode.

Following any processing of voice data clips, the message is passed to the HARSlotManager to download the clips to the appropriate slot on the HAR device using the ISSAP55HAR object. The HARSlotManager keeps track of all slots in use on the HAR controller, including the clips that occupy the slot and how the slot is being used (Immediate message, default message, etc.) The ISSAP55HAR object is used to carry out the communications to store one or more clips in a slot on the HAR device, including piecing together clips when multiple clips are to be stored in a single slot.

After the HARSlotManager has the clips stored into the HAR controller, a call is made to the ISSAPP55HAR object to have it command the HAR device to play the slot (or slots) that contain the immediate message.

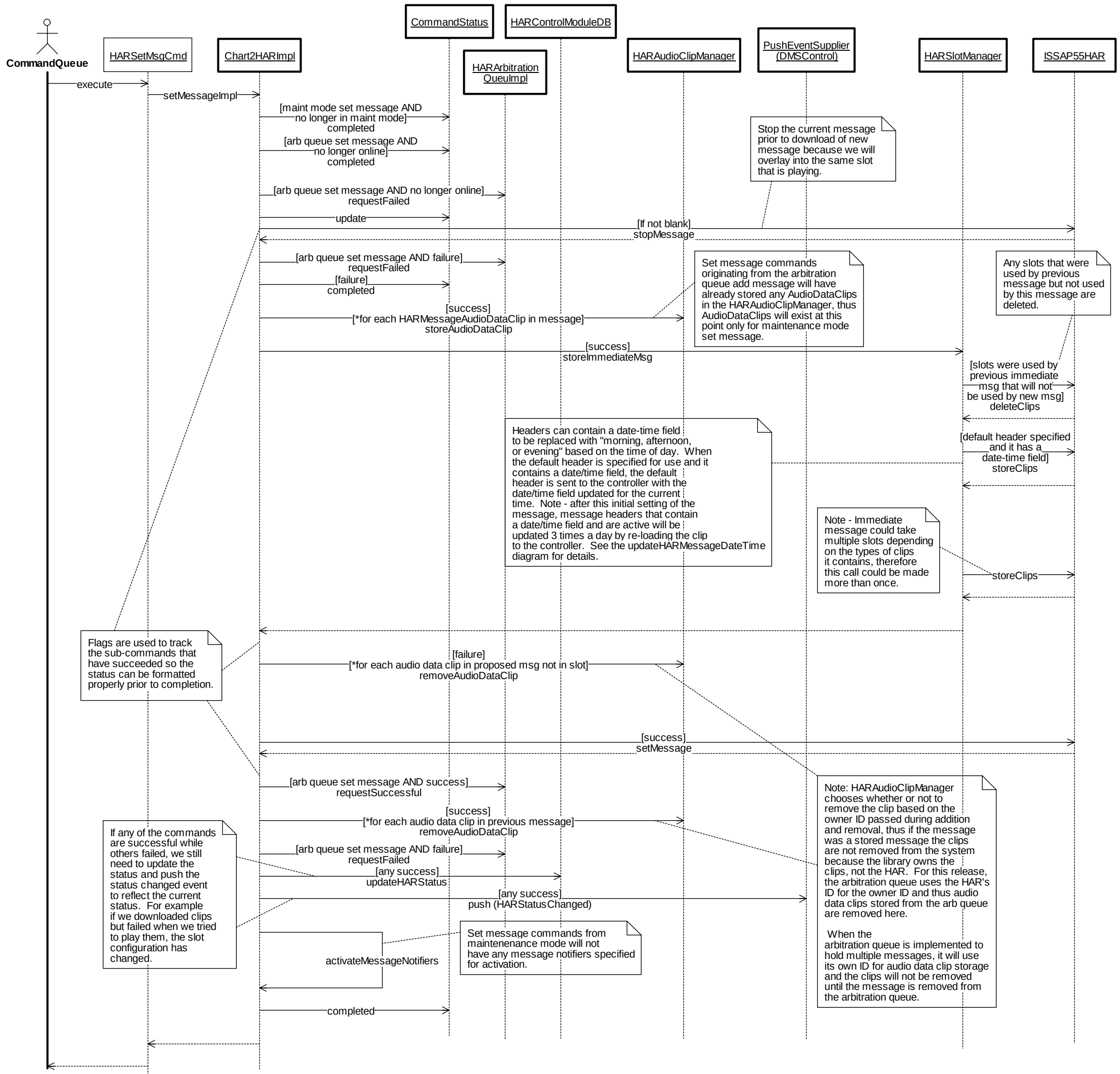

Figure 87. HARControlModule:setMessageImpl (Sequence Diagram)

3.8.2.23 HARControlModule:setTransmitterOff (Sequence Diagram)

A user with proper functional rights can set the HAR transmitter off when the HAR is in maintenance mode. This call is executed asynchronously with the communications being delegated to the ISSAP55HAR class.

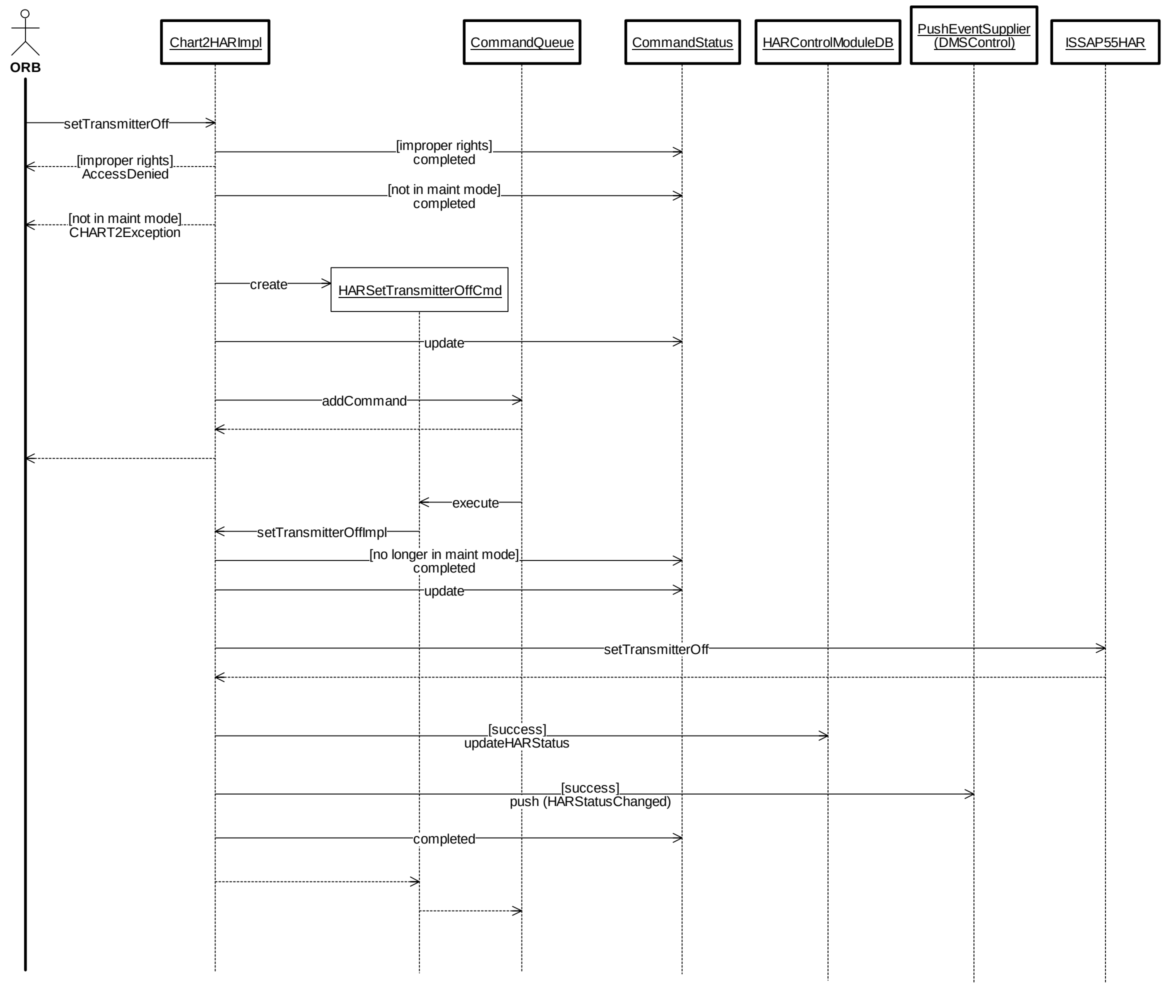

Figure 88. HARControlModule:setTransmitterOff (Sequence Diagram)

3.8.2.24 HARControlModule:setTransmitterOn (Sequence Diagram)

A user with proper functional rights can set the HAR transmitter on when the HAR is in maintenance mode. This call is executed asynchronously with the communications being delegated to the ISSAP55HAR class.

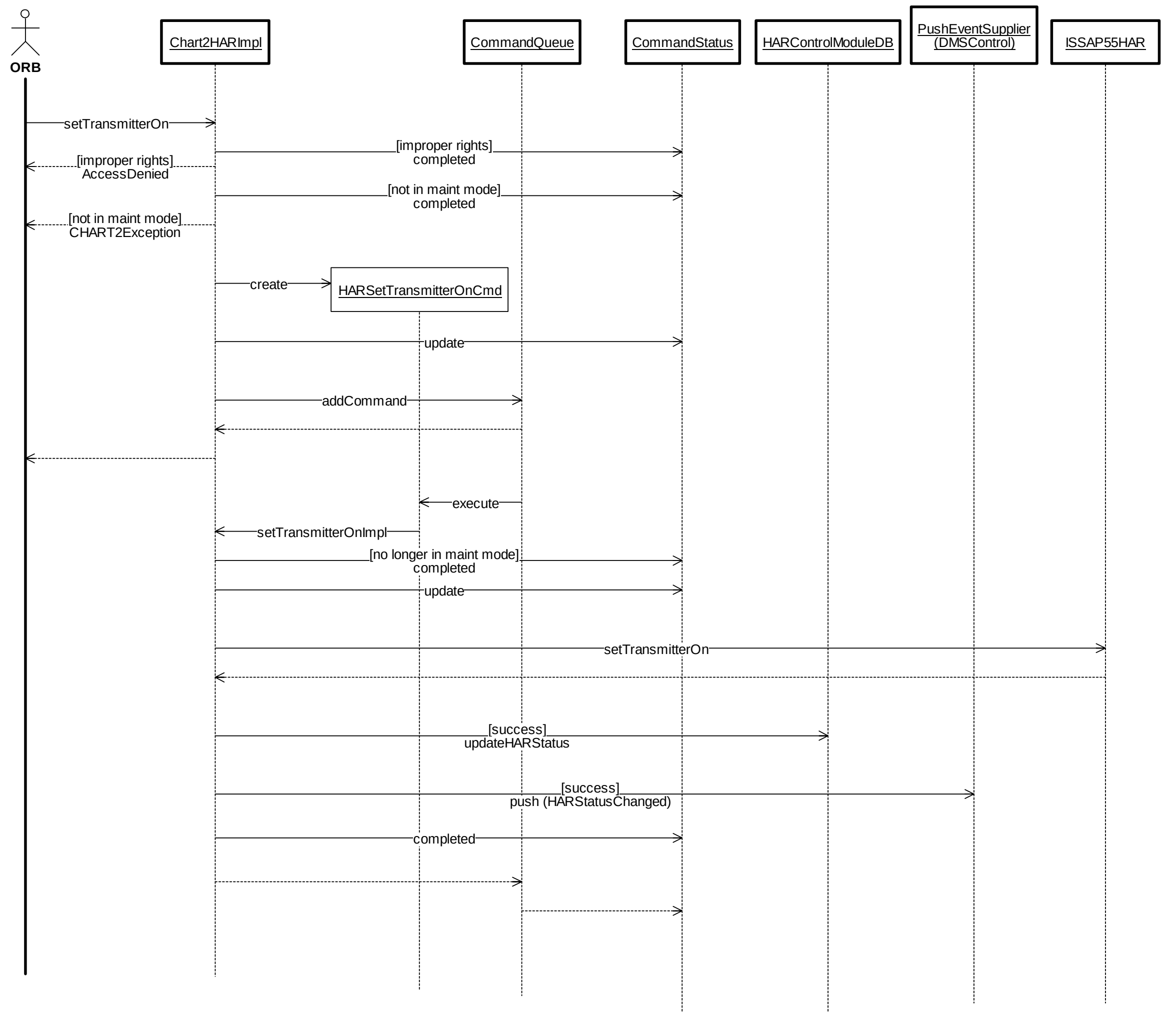

Figure 89. HARControlModule:setTransmitterOn (Sequence Diagram)

3.8.2.25 HARControlModule:setup (Sequence Diagram)

The setup command involves re-sending the current setup (as known in CHART II) to the HAR device. This includes setting the configurable parameters on the HAR, downloading all messages that are to be stored in slots on the HAR, setting the HAR to its default message, and turning the transmitter on. Because this involves many steps, it is possible that only partial success is achieved. In this case, flags are used to keep track of which parts failed and an appropriate status message is relayed to the end-user via the command status object.

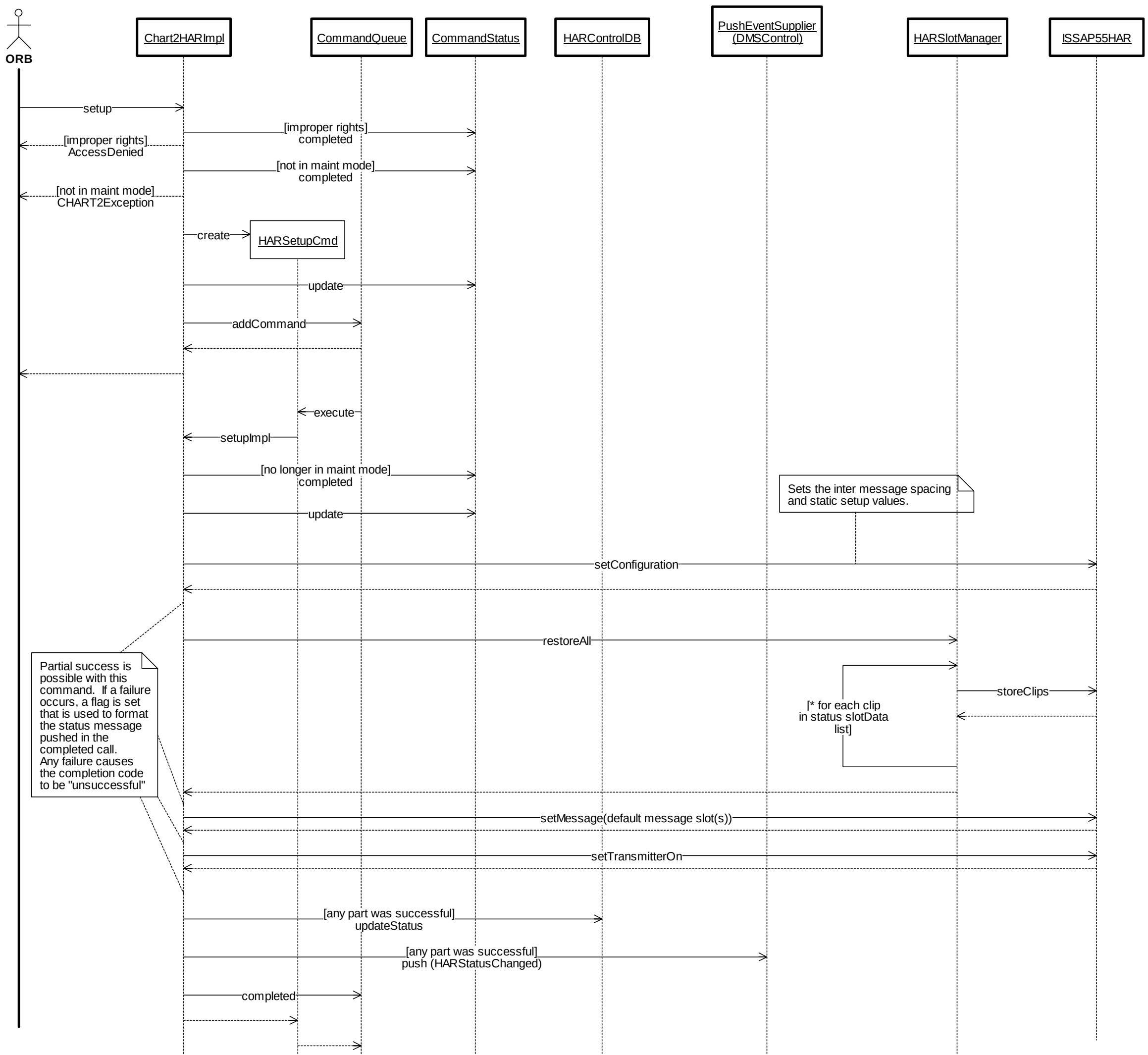

Figure 90. HARControlModule:setup (Sequence Diagram)

3.8.2.26 HARControlModule:storeSlotMessage (Sequence Diagram)

A user with proper functional rights can store a message in a slot in the HAR controller for later activation. This command is processed asynchronously via the command queue. When executed, the HARAudioClipManager object is used to download the message to the HAR and track the slot usage.

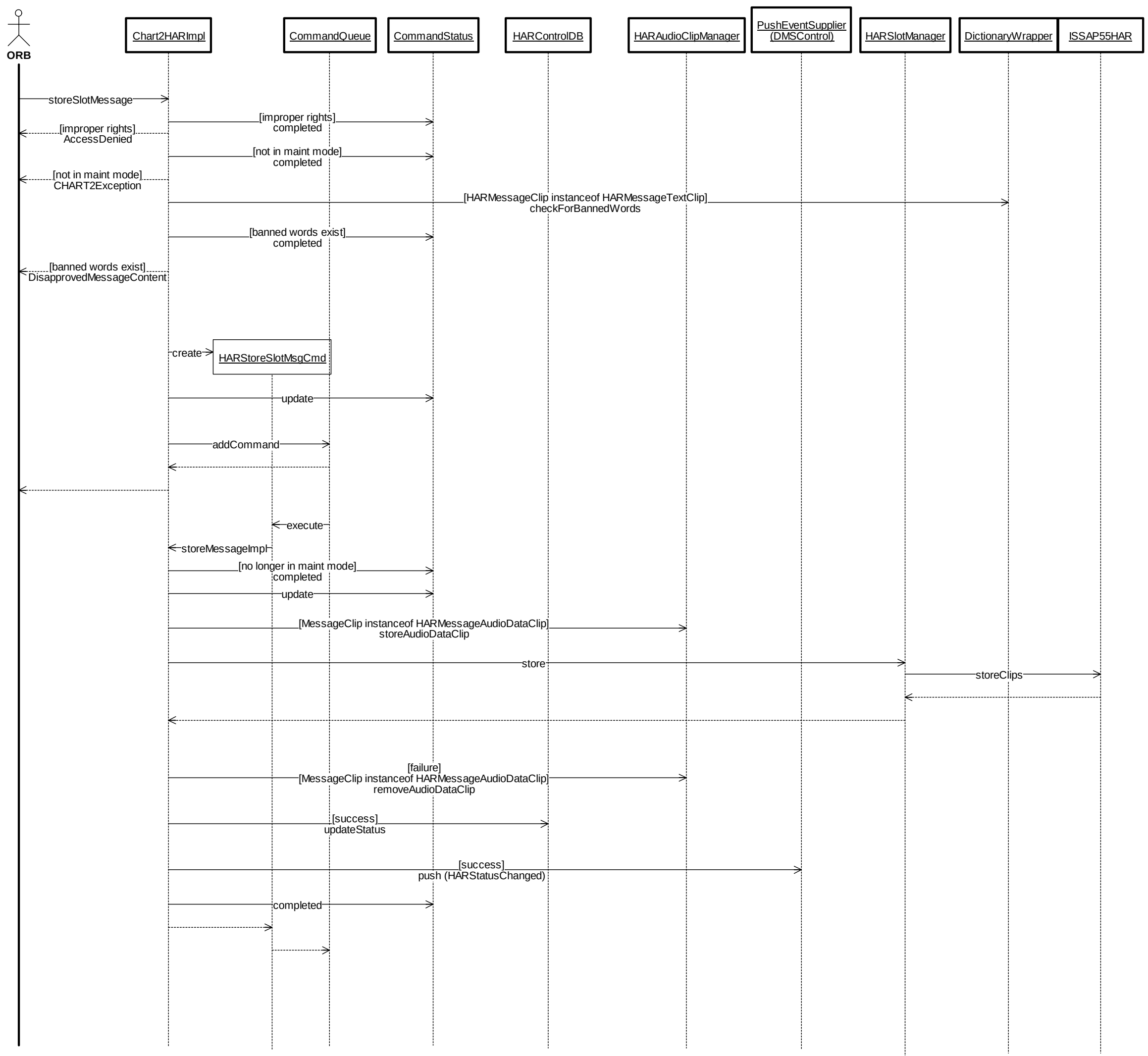

Figure 91. HARControlModule:storeSlotMessage (Sequence Diagram)

3.8.2.27 HARControlModule:TakeOffline (Sequence Diagram)

A user with appropriate priveleges can take a HAR offline. This causes the HAR to be blanked and its transmitter to be set off. If the HAR cannot be blanked, it is still marked as blank within the CHART II system and the device moves to the offline state.

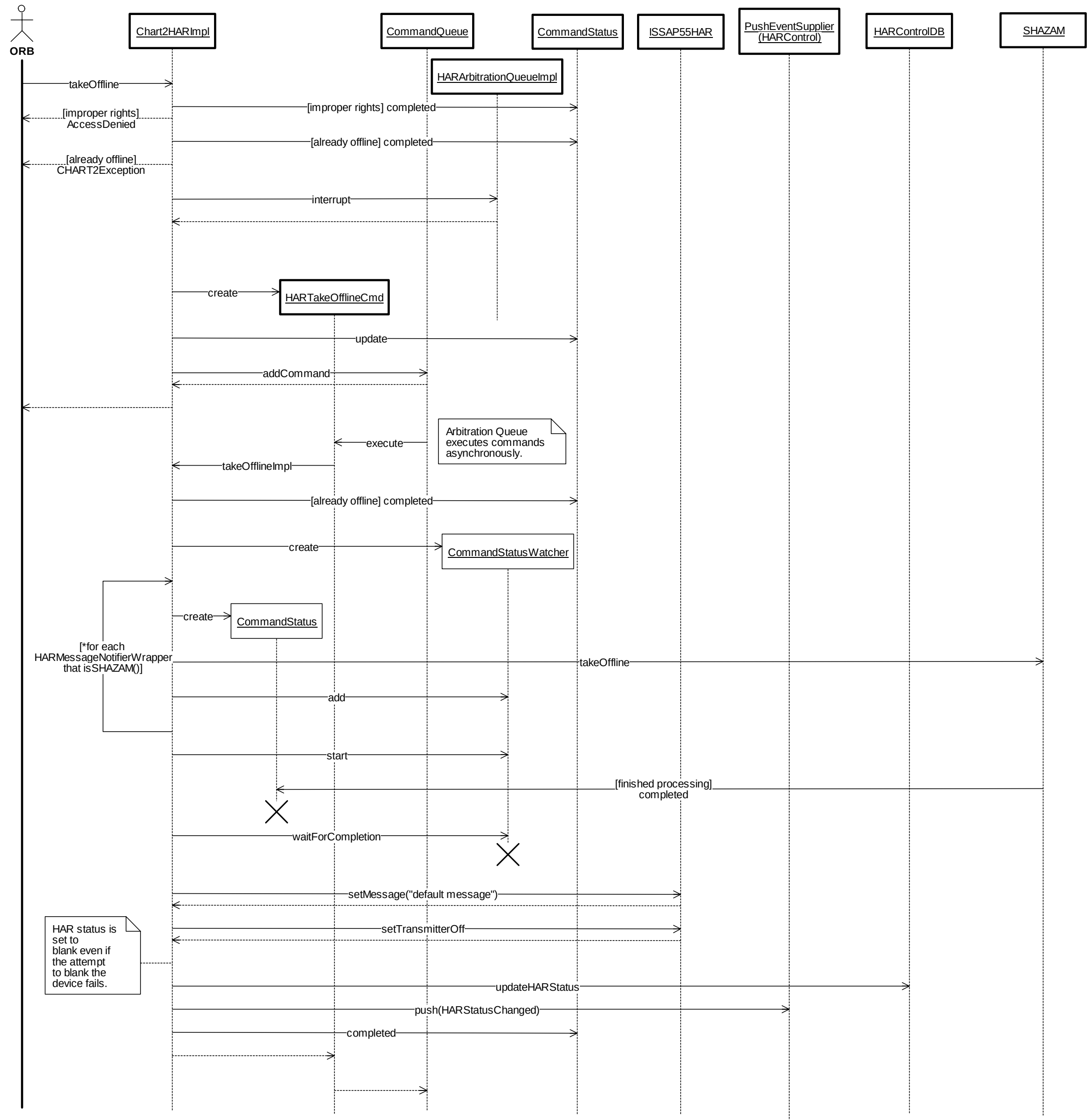

Figure 92. HARControlModule:TakeOffline (Sequence Diagram)

3.8.2.28 HARControlModule:UpdateHARMessageDateTime (Sequence Diagram)

HAR Text messages can contain a tag that is to be substituted with the text “morning”, “afternoon”, or “evening” in place of the tag based on the time of day the message is set. This substitution will be done at the time the HAR message is set and will also be done to any messages that are active at 00:00, 12:00, and 17:00. This sequence diagram shows the processing involved in the automated substitution and message setting. This automated process involves telling each HAR object to update its message if it deems it is necessary. If necessary, the HAR puts a command on its command queue and the command is executed asynchronously. Because the command queue may have had a command in progress that changes the HAR’s message, it is necessary to check if the date/time update needs to be done when the command is executed. If so, the appropriate clip (or clips) are re-downloaded to the HAR and the appropriate spoken word will replace the date/time field during the download process, which involves text to speech conversion.

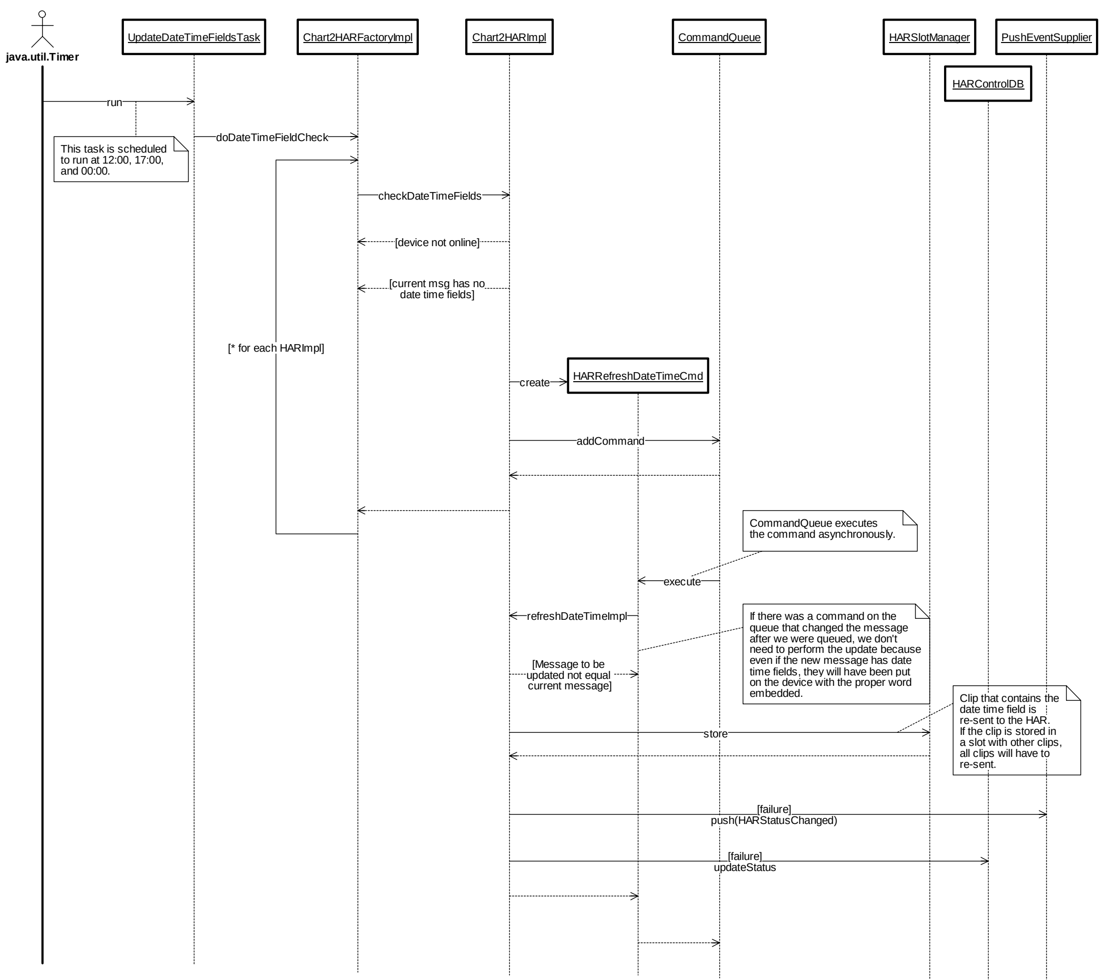

Figure 93. HARControlModule:UpdateHARMessageDateTime (Sequence Diagram)

3.9 HARUtility

3.9.1 Classes

3.9.1.1 HARUtility (Class Diagram)

This class diagram shows classes related to the HAR that are used by both the GUI and the server. Most (if not all) of these classes are implementations of value type classes defined in the system interfaces (IDL).

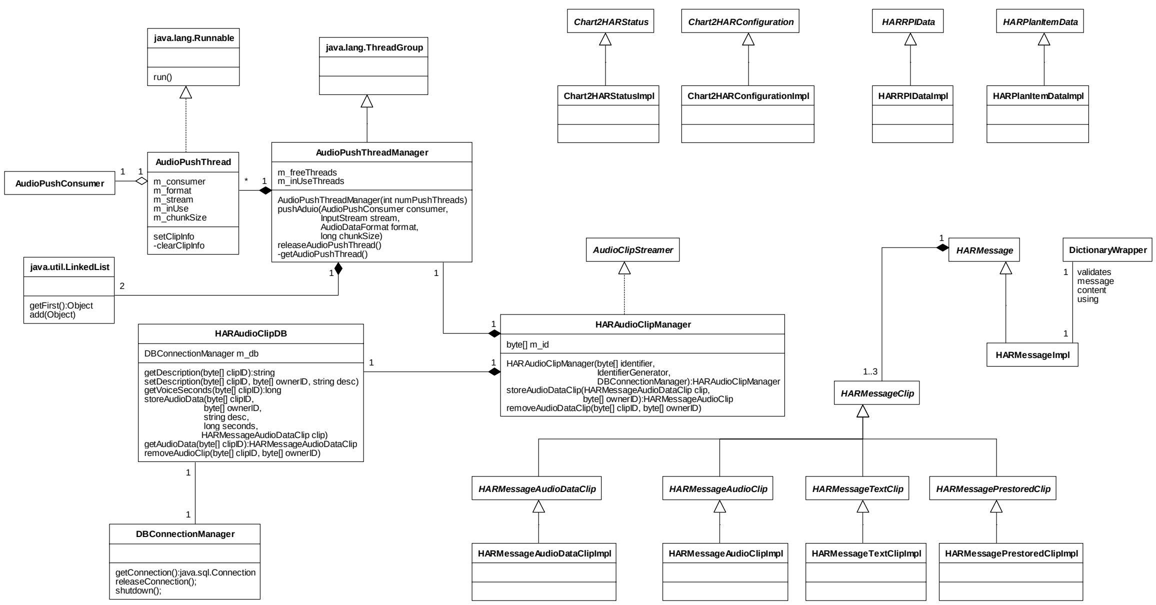

Figure 94. HARUtility (Class Diagram)

3.9.1.1.1 AudioClipStreamer (Class)

This interface is implemented by objects that can push a previously stored audio clip given its ID. The audio data is pushed via the AudioPushConsumer supplied by the user of this interface

3.9.1.1.2 AudioPushConsumer (Class)

This interface is implemented by objects that are capable of receiving audio data using the push model, where the server pushes the data to the consumer. One call to pushAudioProperties() will always precede any calls to pushAudio().

3.9.1.1.3 AudioPushThread (Class)

This class is a thread that is used to push audio clip information to an AudioPushConsumer.

3.9.1.1.4 AudioPushThreadManager (Class)

This class maintains a pool of reusable AudioPushThread objects, which can be used to push audio clip information back to the client. It provides the functionality to manage access to the AudioPushThreads.

3.9.1.1.5 CHART2HARConfiguration (Class)

This class contains configuration data for the HAR that is used for CHART II specific processing (as opposed to the configuration values contained in HARConfiguration that relate to typical HAR usage).

3.9.1.1.6 CHART2HARConfigurationImpl (Class)

This class is a concrete implementation of the CHART2HARConfiguration abstract class generated from IDL.

3.9.1.1.7 CHART2HARStatus (Class)

This class contains status information for a CHART2HAR object. This information is specific to CHART II processing and extends beyond the status related to typical HAR device control.

3.9.1.1.8 CHART2HARStatusImpl (Class)

This class is a concrete implementation of the CHART2HARStatus abstract class generated from IDL.

3.9.1.1.9 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two separate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the “jdbc.drivers” system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.9.1.1.10 DictionaryWrapper (Class)

This singleton class provides a wrapper for the system dictionary that provides automatic location of the dictionary and automatic re-discovery should the dictionary reference return an error. This class also allows for built-in fault tolerence by automatically failing over to a “working” dictionary without the user of this class being aware that this being done. In addition, this class defers the discovery of the Dictionary until its first use, thus eliminating a start-up dependency for modules that use the dictionary.

This class delegates all of its method calls to the system dictionary using its currently known good reference to the system dictionary. If the current reference returns a CORBA failure in the delegated call, this class automatically switches to another reference. When there are no good references (as is true the first time the object is used), this class issues a trader query to (re)discover the published Dictionary objects in the system. During a method call, the trader will be queried at most one time and under normal circumstances (other than the first use) the trader will not be queried at all.

3.9.1.1.11 HARAudioClipDB (Class)

This class provides access to the database for the HARAudioClipManager. It provides a means to store and retrieve recorded voice to/from the database.

3.9.1.1.12 HARAudioClipManager (Class)

This class provides the implementation of the AudioStreamer interface and is capable of streaming recorded audio clips that have been previously stored. When requested to stream an audio clip, this class pulls the audio data from its persistent store pushes the audio data to the given AudioPushConsumer in a worker thread. This class also allows newly recorded audio clips to be added to the system. When a clip is added to the system it is assigned a unique ID and a HARMessageAudioClip is created as a thin wrapper to provide access to the audio data. When new audio clips are added to the system, the ID of the owner is passed to facilitate clean up of the clip when it is no longer needed.

3.9.1.1.13 HARMessage (Class)

This utility class represents a message that is capable of being stored on a HAR. It stores the HAR message as a HAR message header, body and footer. It contains methods to input and output them in different formats.

3.9.1.1.14 HARMessageAudioClip (Class)

This class is a thin wrapper for recorded voice that is to be played on a HAR. This class is passed around the system instead of passing the actual voice data. When the actual voice data is needed to play to the user or to program the HAR device, this object’s streamer is used to stream the actual voice data.

3.9.1.1.15 HARMessageAudioClipImpl (Class)

This class defines HARMessageAudioClip as defined in the IDL. Refer to HARMessageAudioClip for details.

3.9.1.1.16 HARMessageAudioDataClip (Class)

This class is a message clip that contains audio data and the format of the audio data. Because audio data can be very large, this type of clip is reserved for use when recorded voice is first entered into the system. Recorded voice that already exists in the system is passed throughout the system using HARMessageAudioClip to avoid sending the large audio data when possible.

3.9.1.1.17 HARMessageAudioDataClipImpl (Class)

This class implements the HARMessageAudioDataClip as defined in the IDL. Refer to HARMessageAudioDataClip for details.

3.9.1.1.18 HARMessageClip (Class)

This class represents a section of a HAR message. It can be either plain text that would need to be converted to audio prior to broadcast, or binary format (MP3, WAV, etc.)

3.9.1.1.19 HARMessageImpl (Class)

This class is a concrete implementation of the HARMessage abstract class generated from IDL.

3.9.1.1.20 HARMessagePrestoredClip (Class)

This class stores data used to identify the usage of a clip that has already been stored on a HAR device.

3.9.1.1.21 HARMessagePrestoredClipImpl (Class)

This class implements HARMessagePrestoredClip as defined in IDL. Refer to HARMessagePrestoredClip for details.

3.9.1.1.22 HARMessageTextClip (Class)

This class represents a HAR message content object that is in plain text format. This message can be checked for banned words and will be converted into a voice message using a speech engine to broadcast on a HAR device.

3.9.1.1.23 HARMessageTextClipImpl (Class)

This class implements HARMessageTextClip as defined in the IDL. Refer to HARMessageTextClip for details.

3.9.1.1.24 HARPlanItemData (Class)

This class is used to associate a message with a HAR for use in Plans.

3.9.1.1.25 HARPlanItemDataImpl (Class)

This class is a concrete implementation of the HARPlanItemData abstract class generated from IDL.

3.9.1.1.26 HARRPIData (Class)

This class represents an item in a traffic event response plan that is capable of issuing a command to put a message on a HAR when executed. When the item is executed, it adds the message to the arbitration queue of the specified HAR. When the item is removed from the response plan (manually or implicitly through closing the traffic event) the item asks the HAR's arbitration queue to remove the message.

3.9.1.1.27 HARRPIDataImpl (Class)

This class is a concrete implementation of the HARRPIData abstract class generated from IDL.

3.9.1.1.28 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

3.9.1.1.29 java.lang.ThreadGroup (Class)

A thread group represents a set of threads.

3.9.1.1.30 java.util.LinkedList (Class)

This class is an implementation of List interface for a linked list.

3.9.2 Sequence Diagrams

3.9.2.1 HARUtility:PushAudio (Sequence Diagram)

This diagram shows how audio data is pushed back to the client. The AudioPushThreadManager manages a pool of threads that can be used to push audio data back to the clients. When a request is made to push audio, the AudioPushThreadManager looks in the thread list for a free thread. If all the threads are being used, the request waits until a thread becomes available. Once a thread becomes available, the thread is notified of the clip by setting the clip data and the thread starts pushing the audio data by first pushing the audio properties. Then, the thread starts to push the audio data in chunks of the size requested by the client. If the pushing operation fails, an error is passed to the consumer. At the completion of pushing, the thread clears the clip data and informs the AudioPushThreadManager to free the thread. The AudioPushThreadManager in turn frees the thread and notifies any waiting request.

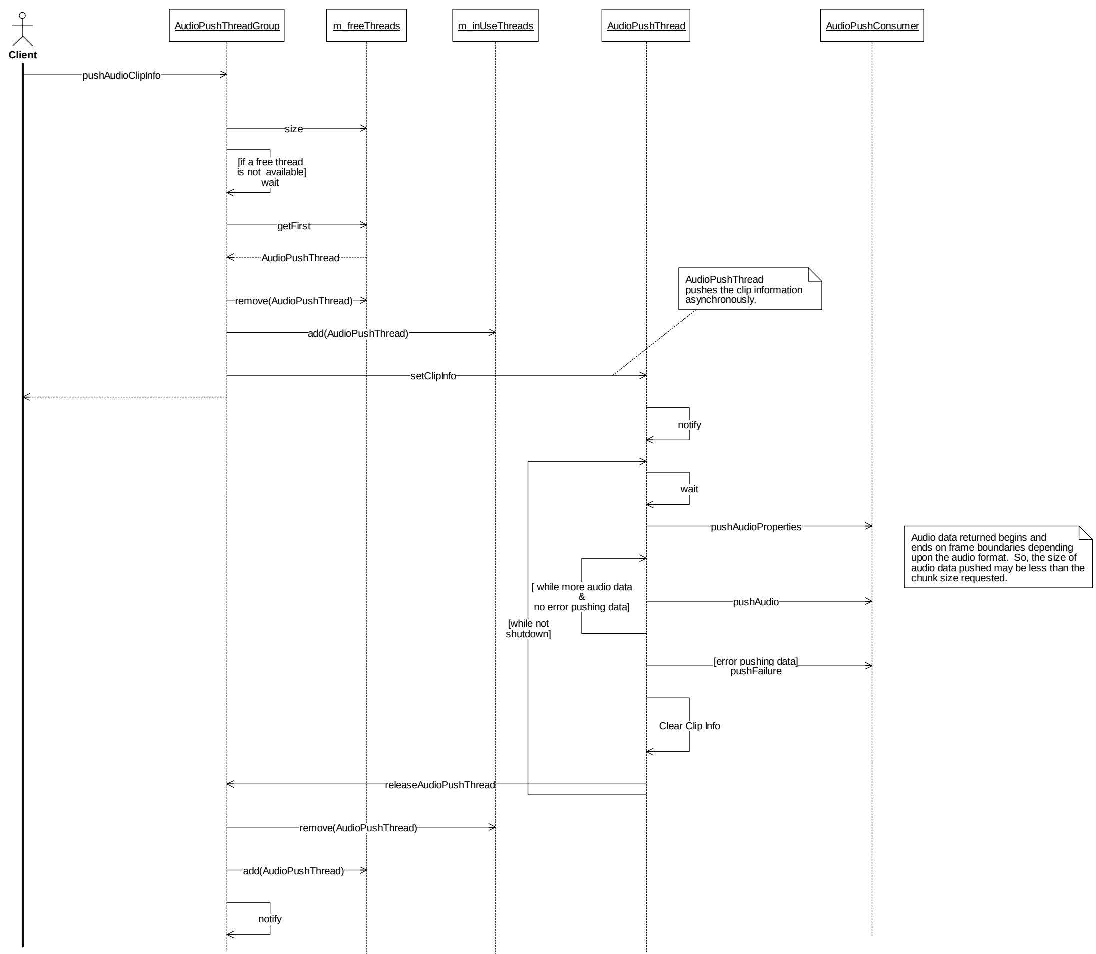

Figure 95. HARUtility:PushAudio (Sequence Diagram)

3.9.2.2 HARUtility:StoreAudioClip (Sequence Diagram)

When a CHART2HARImpl or the MessageLibraryDB object have been passed a HAR message that contains a HARMessageAudioDataClip, the HARAudioClipManager is called to store the voice data and create a thin wrapper object that represents the voice data. This thin wrapper is passed around the system instead of the voice data itself. The thin wrapper contains a reference to the HARAudioClipManager which will push the voice data to any holders of the thin wrapper that request the actual voice data.

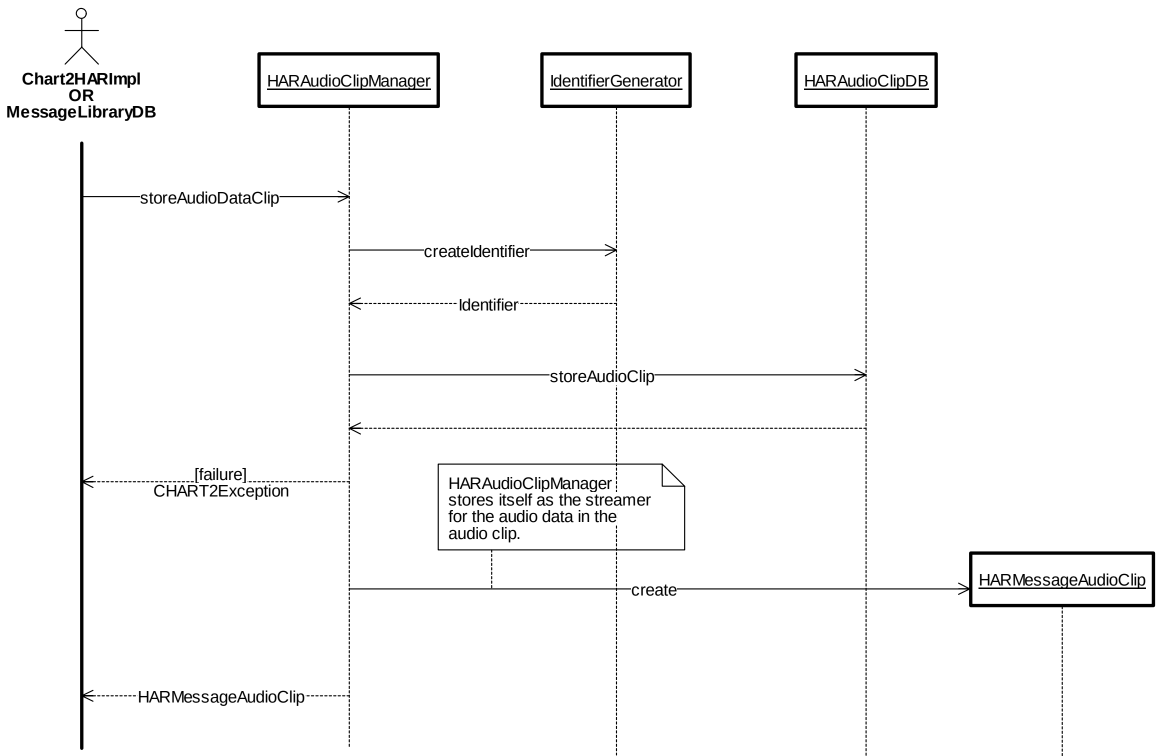

Figure 96. HARUtility:StoreAudioClip (Sequence Diagram)

3.10 JavaClasses

3.10.1 Classes

3.10.1.1 JavaClasses (Class Diagram)

This package is included for reference to classes included in the Java programming language that are used in class and sequence diagrams for other packages within this design.

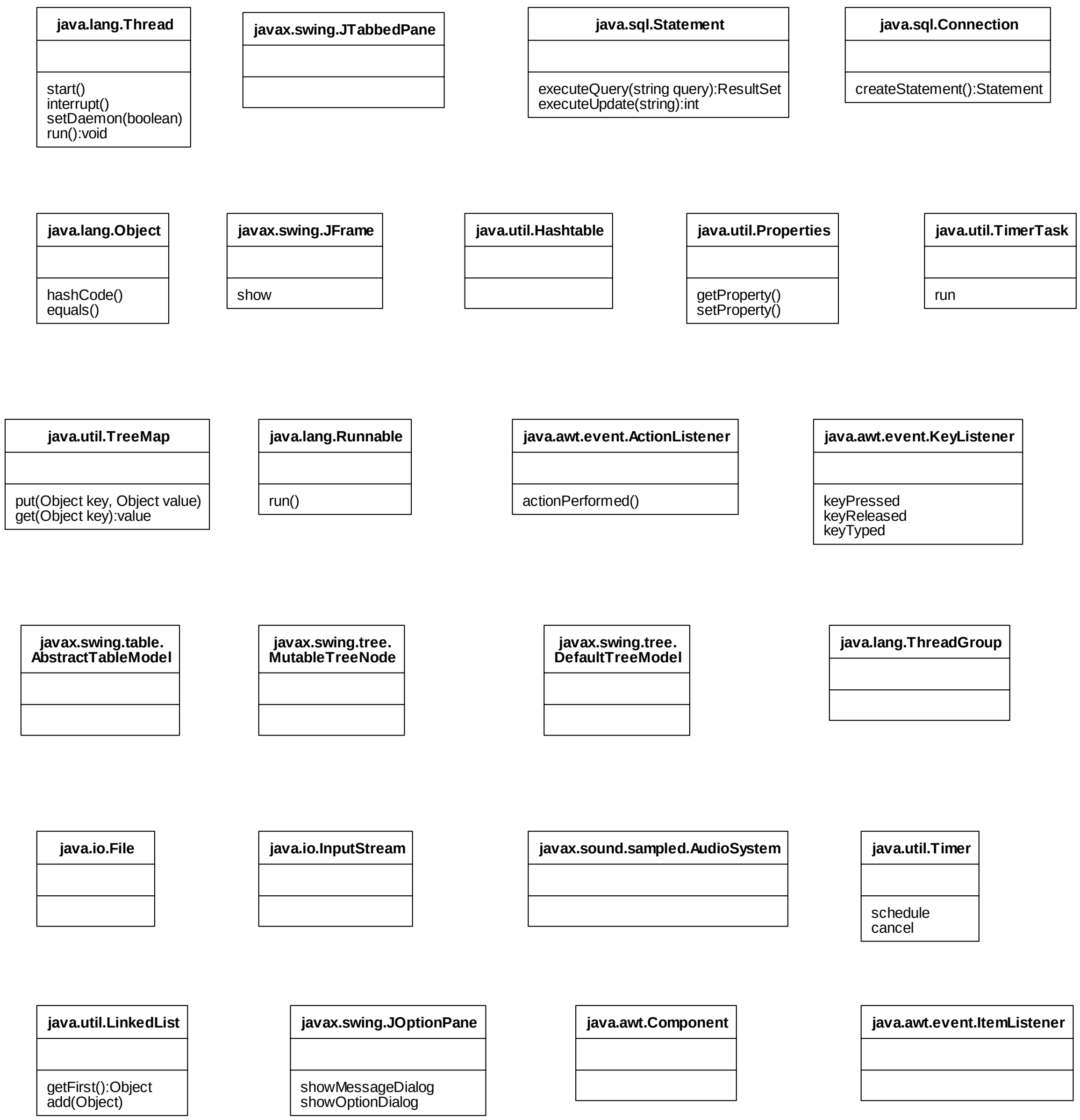

Figure 97. JavaClasses (Class Diagram)

3.10.1.1.1 java.awt.Component (Class)

This class is the base class for all graphical user interface components such as buttons and panels.

3.10.1.1.2 java.awt.event.ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

3.10.1.1.3 java.awt.event.ItemListener (Class)

This interface allows the implementing class to listen for changes to an item such as a list item or combo box item.

3.10.1.1.4 java.awt.event.KeyListener (Class)

Interface that a class must realize in order for objects of that class to be notified when the user presses a key.

3.10.1.1.5 java.io.File (Class)

This class is an abstract representation of file and directory pathnames.

3.10.1.1.6 java.io.InputStream (Class)

Java class that represents a input stream of bytes.

3.10.1.1.7 java.lang.Object (Class)

This is the base class from which all Java classes inherit.

3.10.1.1.8 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

3.10.1.1.9 java.lang.Thread (Class)

This class represents a java thread of execution.

3.10.1.1.10 java.lang.ThreadGroup (Class)

A thread group represents a set of threads.

3.10.1.1.11 java.sql.Connection (Class)

This class represents a connection (session) with a specific database.

3.10.1.1.12 java.sql.Statement (Class)

Java class used for executing a static SQL statement and obtaining the results produced by it.

3.10.1.1.13 java.util.Hashtable (Class)

This class implements a hashtable, which is a data structure that maps keys to values. Any non-null object can be used as a key or as a value. Objects used as keys implement the hashCode method that is inherited by all objects from the java.lang.Object class.

3.10.1.1.14 java.util.LinkedList (Class)

This class is an implementation of List interface for a linked list.

3.10.1.1.15 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its “defaults”; this second property list is searched if the property key is not found in the original property list.

3.10.1.1.16 java.util.Timer (Class)

This class provides asynchronous execution of tasks that are scheduled for one-time or recurring execution.

3.10.1.1.17 java.util.TimerTask (Class)

This class is an abstract base class which can be scheduled with a timer to be executed one or more times.

3.10.1.1.18 java.util.TreeMap (Class)

This class is an implementation of the SortedMap interface. This class guarantees that the map will be in ascending key order, sorted according to the natural order for the key’s class, or by the comparator provided at creation time, depending on which constructor is used.

3.10.1.1.19 javax.sound.sampled.AudioSystem (Class)

The AudioSystem class acts as the entry point to the sampled-audio system resources. This class lets you query and access the mixers that are installed on the system.

3.10.1.1.20 javax.swing.JFrame (Class)

Java class that displays a frame window.

3.10.1.1.21 javax.swing.JOptionPane (Class)

This class is used to display popup messages to an end user.

3.10.1.1.22 javax.swing.JTabbedPane (Class)

This class is a component that has tabbed pages, and the user can click on a tab to flip to a certain page.

3.10.1.1.23 javax.swing.table. AbstractTableModel (Class)

This class provides a base implementation of the TableModel interface. This data structure will be used to supply a JTable with data.

3.10.1.1.24 javax.swing.tree. DefaultTreeModel (Class)

This class is the data structure that is used as a foundation for the JTree class.

3.10.1.1.25 javax.swing.tree. MutableTreeNode (Class)

This interface extends the TreeNode interface and provides the ability to add and remove children from nodes. It may be used in a TreeModel.

3.11 MessageLibraryModule

3.11.1 Classes

3.11.1.1 MessageLibraryModuleClasses (Class Diagram)

The MessageLibraryModule is a Service Application module that serves the MessageLibraryFactory, MessageLibrary and StoredMessage objects to the rest of the CHART2 system. This diagram shows how the implementation of these CORBA interfaces rely on other supporting classes to perform their functions.

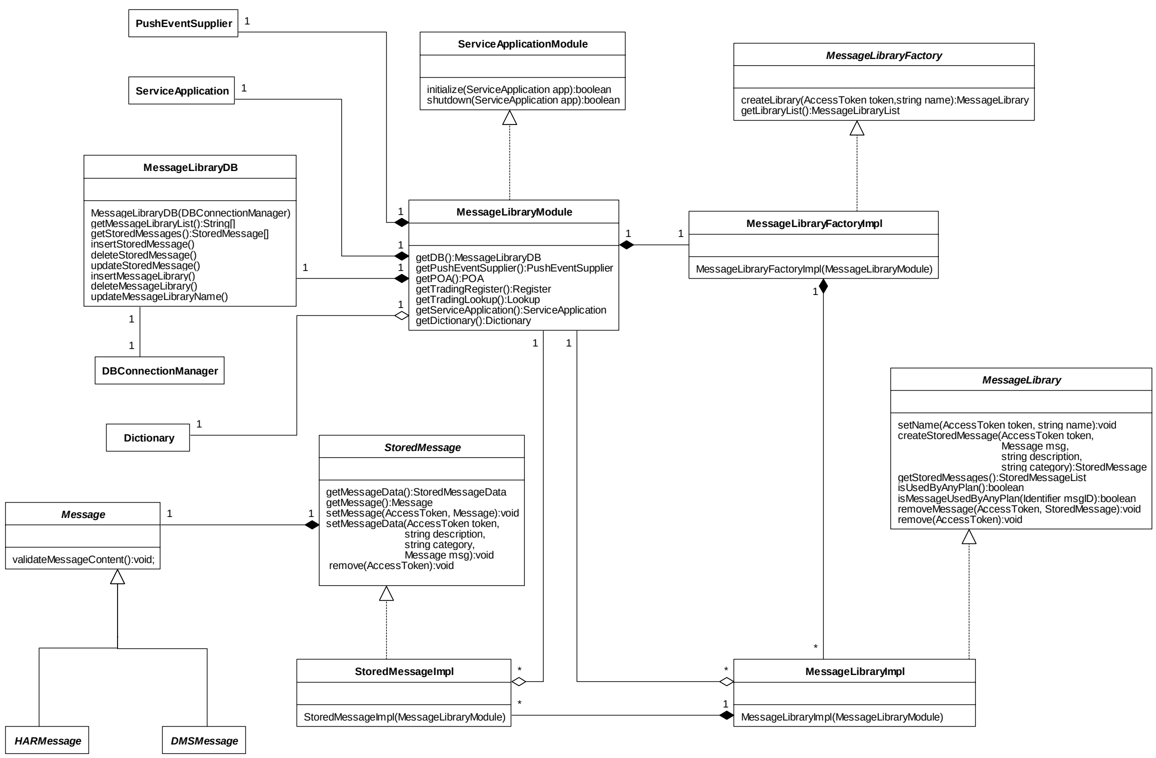

Figure 98. MessageLibraryModuleClasses (Class Diagram)

3.11.1.1.1 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two separate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the “jdbc.drivers” system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.11.1.1.2 Dictionary (Class)

The Dictionary IDL interface provides functionality to add, delete and check for words that are approved or banned from being used in a CHART2 messaging device. Examples of messaging devices are DMS, HAR etc.

3.11.1.1.3 DMSMessage (Class)

The DMSMessage class is an abstract class that describes a message for a DMS. It consists of two elements: a MULTI-formatted message and beacon state information (whether the message requires that the beacons be on). The DMSMessage is contained within a DMSStatus object, used to communicate the current message on a sign, and is stored within a DMSRPIData object, used to specify the message that should be on a sign when the response plan item is executed.

3.11.1.1.4 HARMessage (Class)

This utility class represents a message capable of being stored on a HAR. It stores the HAR message as a HAR message header, body and footer. It contains methods to input and output them in different formats.

3.11.1.1.5 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.11.1.1.6 MessageLibrary (Class)

This class represents a logical collection of messages that are stored in the database.

3.11.1.1.7 MessageLibraryFactory (Class)

This class is used to create new message libraries and maintain them in a collection.

3.11.1.1.8 MessageLibraryFactoryImpl (Class)

The MessageLibraryFactoryImpl class provides an implementation of the MessageLibraryFactory interface as defined in the IDL. The MessageLibraryFactory maintains a list of MessageLibraryImpl objects and is responsible for publishing MessageLibrary objects in the Trader.

3.11.1.1.9 MessageLibraryDB (Class)

The MessageLibraryDB class is a collection of methods that perform database operations on tables pertinent to Message Library Management. The class is constructed with a Connection Manager object, which manages database connections. Every operation in this class obtains a connection to the database from the connection manager prior to performing the requested DB operation.

3.11.1.1.10 MessageLibraryImpl (Class)

The MessageLibraryImpl class provides an implementation of the MessageLibrary interface as specified in the IDL. The MessageLibrary maintains a list of StoredMessage objects and is responsible for publishing StoredMessage objects in the Trader.

3.11.1.1.11 MessageLibraryModule (Class)

This class implements the ServiceApplicationModule interface. It creates and serves a single MessageLibraryFactoryImpl object, which in turn serves MessageLibraryImpl objects. This module also serves StoredMessage objects that were created in the message libraries being served by this module.

3.11.1.1.12 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier’s push rate.

3.11.1.1.13 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.11.1.1.14 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.11.1.1.15 StoredMessage (Class)

This class holds a message object that is stored in a message in a library. It contains attributes such as category and message description which are used to allow the user to organize messages.

3.11.1.1.16 StoredMessageImpl (Class)

The StoredMessageImpl class provides an implementation of the StoredMessage interface as specified in the IDL.

3.11.2 Sequence Diagrams

3.11.2.1 MessageLibraryModule:CreateDMSStoredMessage (Sequence Diagram)

An operator with the correct functional rights may create a stored message for display on a DMS device. The GUI will create a Message object based on the type of stored message the user would like to create. In this case, a DMSMessage object is created. The message library is called to create a stored message. The message library will check if the user has the appropriate rights. If they do, the message will be checked for banned words. If the message contains banned words, an error is returned. If not, a stored message is created, the newly created stored message data is inserted into the database and the stored message object will be published in the CORBA trading service and other system components will be notified of its existence via the CORBA event service. Note that even though a dictionary check is done at the time of storage, the dictionary is always checked on the server side prior to allowing a message to be set on a DMS. The user and operation details are logged in the operations log.

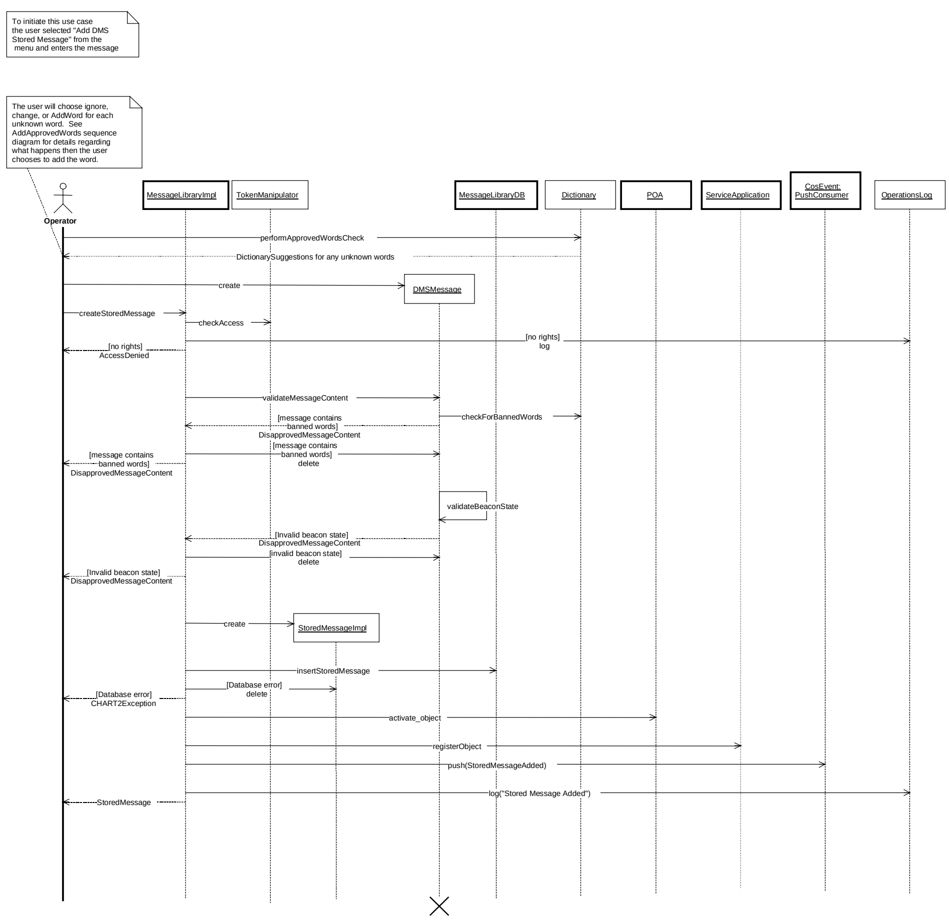

Figure 99. MessageLibraryModule:CreateDMSStoredMessage (Sequence Diagram)

3.11.2.2 MessageLibraryModule:CreateHARStoredMessage (Sequence Diagram)

An operator with the correct functional rights may create a stored message for use on a HAR device. The GUI will create a Message object based on the type of stored message the user would like to create. In this case, a HARMessage object is created. A HARMessage consists of three HAR message clips that can either be in binary or text format. The message library is called to create a stored message. The message library will check if the user has the appropriate rights. If they do, the message is validated by calling the Dictionary to check for disapproved words. Note that only the clips that are in text format will be checked for banned words. If the message contains banned words, an error is returned. If not, a stored message is created, the newly created stored message data is inserted into the database and the stored message object will be published in the CORBA trading service and other system components will be notified of its existence via the CORBA event service. Note that even though a dictionary check is done at the time of storage, the dictionary is always checked on the server side prior to downloading the message to the HAR. The user and operation details are logged in the operations log.

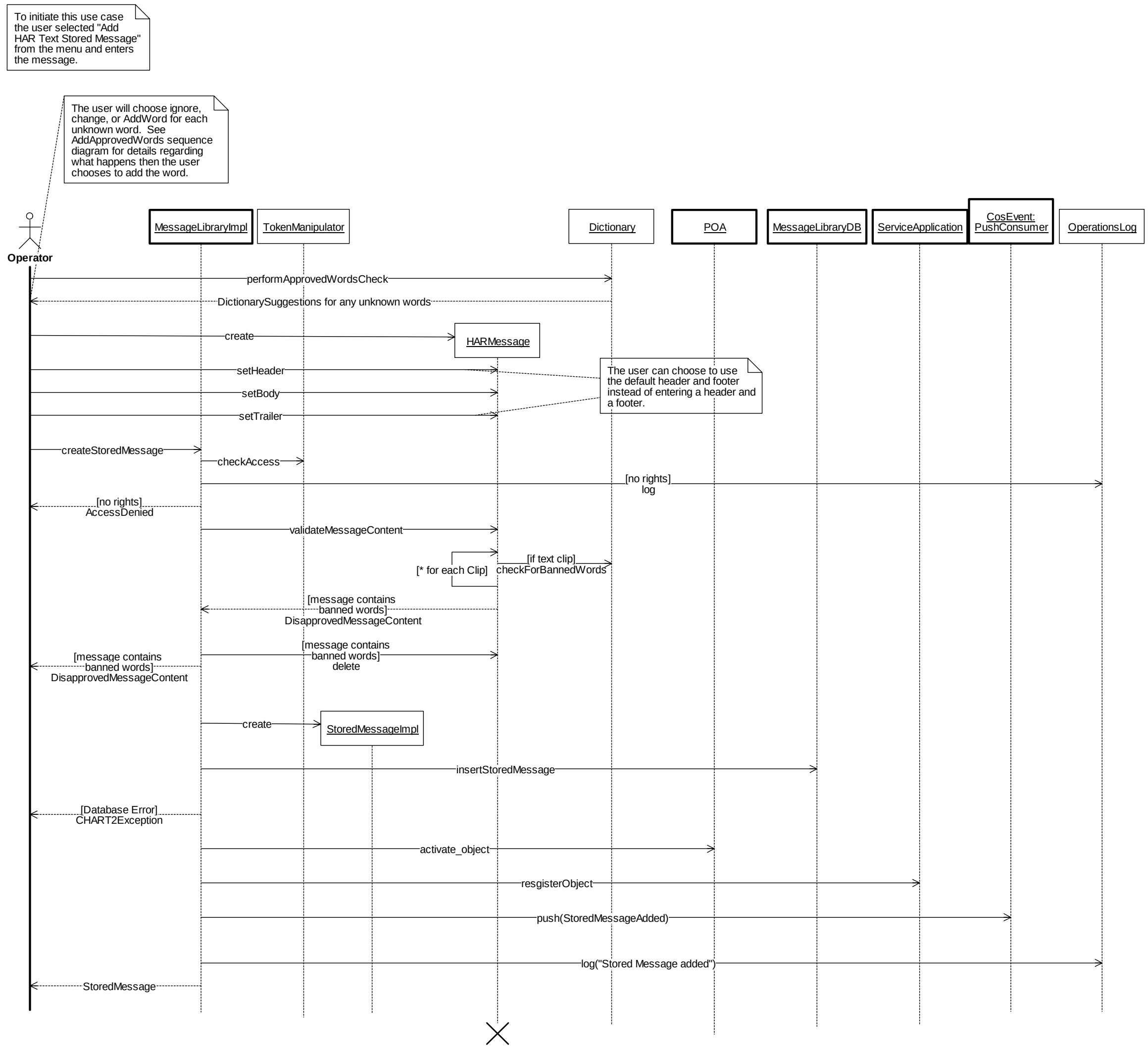

Figure 100. MessageLibraryModule:CreateHARStoredMessage (Sequence Diagram)

3.11.2.3 MessageLibraryModule:CreateMessageLibrary (Sequence Diagram)

A user possessing the proper functional rights can add a Message Library to the system. The library object is created and published via the CORBA Trading Service. An event is pushed via the CORBA Event Service to notify interested parties of the new library. The user and operation details are logged in the operations log.

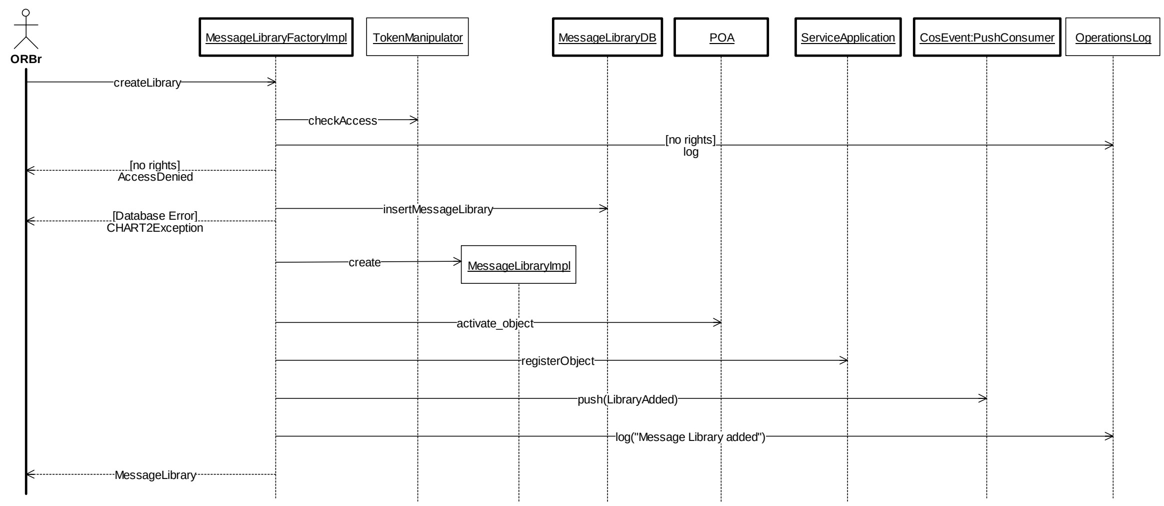

Figure 101. MessageLibraryModule:CreateMessageLibrary (Sequence Diagram)

3.11.2.4 MessageLibraryModule:DeleteMessageLibrary (Sequence Diagram)

A user with the proper functional rights can remove a Message Library from the system. This will include the removal of all stored messages contained within the library. Since stored messages may be used in Plans, a check is made for any plans that may contain the stored messages being deleted and the user is warned. If the user acknowledges the deletions, each message within the library is removed, events are pushed to notify others of the action, and the library is removed from the Trading Service. The user and operation details are logged in the operations log.

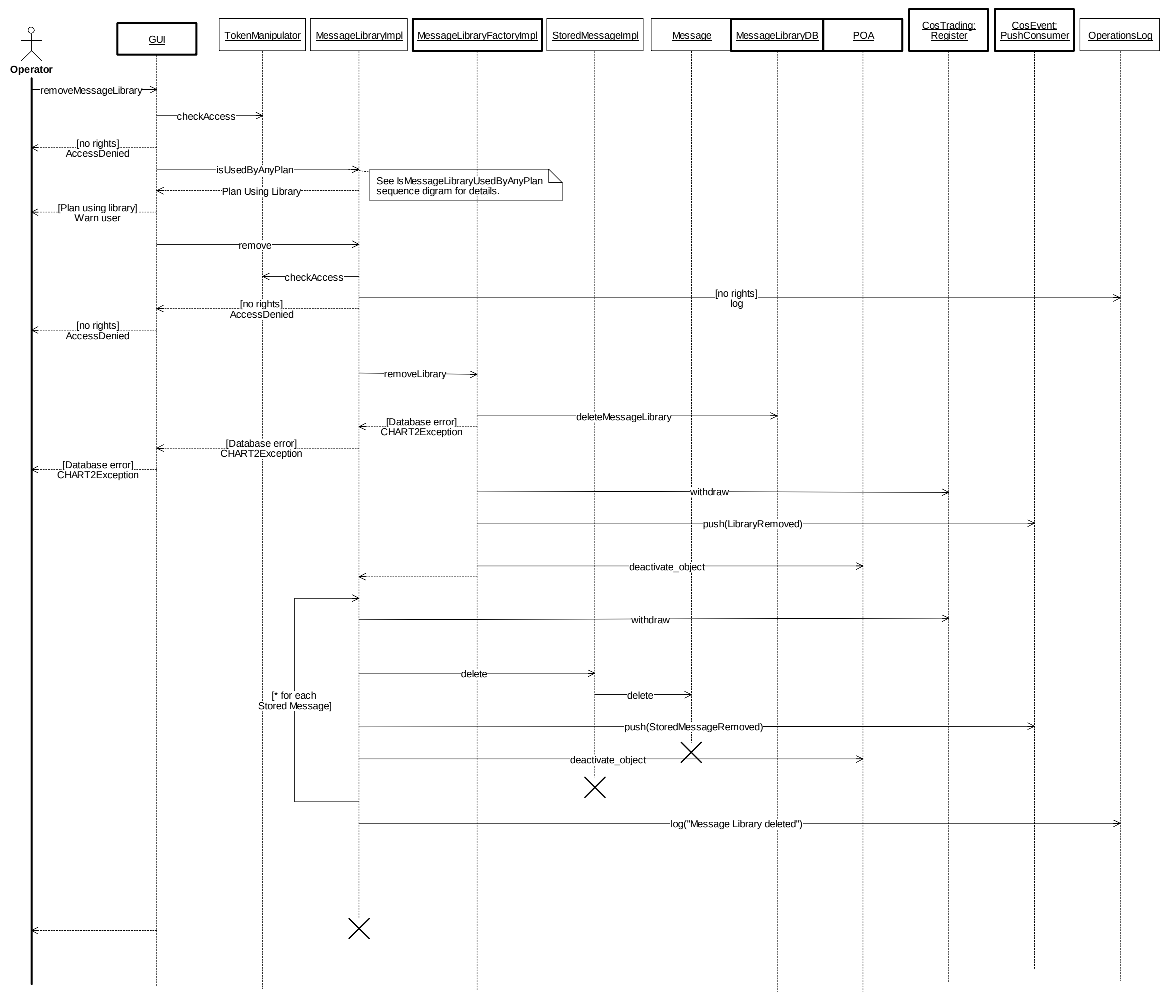

Figure 102. MessageLibraryModule:DeleteMessageLibrary (Sequence Diagram)

3.11.2.5 MessageLibraryModule:DeleteStoredMessage (Sequence Diagram)

A user with the proper functional rights may remove a stored message from the system. Since a stored message may be used in a plan, a check is made to see if the message is used in a plan so that the user can be warned accordingly. The act of deleting the stored message involves deleting the message, updating the database and pushing an event to notify others that the message has been removed from its library. The user and operation details are logged in the operations log.

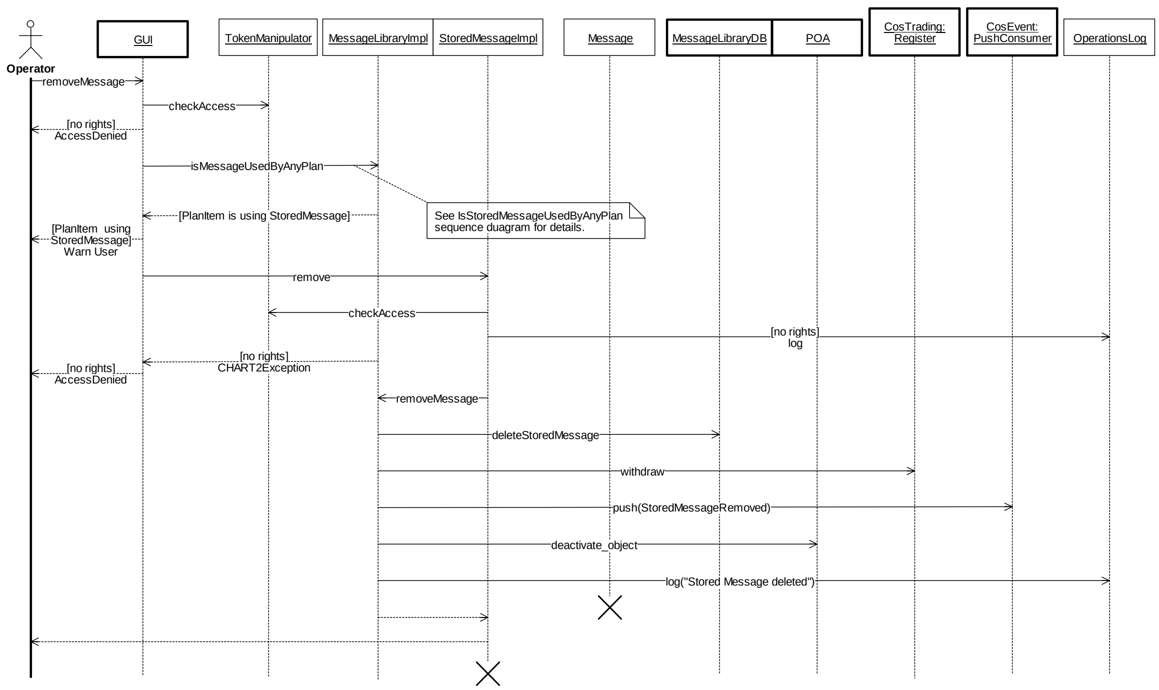

Figure 103. MessageLibraryModule:DeleteStoredMessage (Sequence Diagram)

3.11.2.6 MessageLibraryModule:Initialize (Sequence Diagram)

This sequence diagram shows the startup for the Message Library Module. This module is created by a service that will host this module's objects. A ServiceApplication is passed to this module's initialize method and provides access to basic objects needed by this module. This module creates a Message Library Factory that in turn creates Message Library objects. Message Library objects contain Stored Message objects that are created by the Message Library DB at startup. The MessageLibraryFactory, MessageLibrary and StoredMessage objects are published via the CORBA Trading service to make them available for modifications (given the proper access rights) and usage.

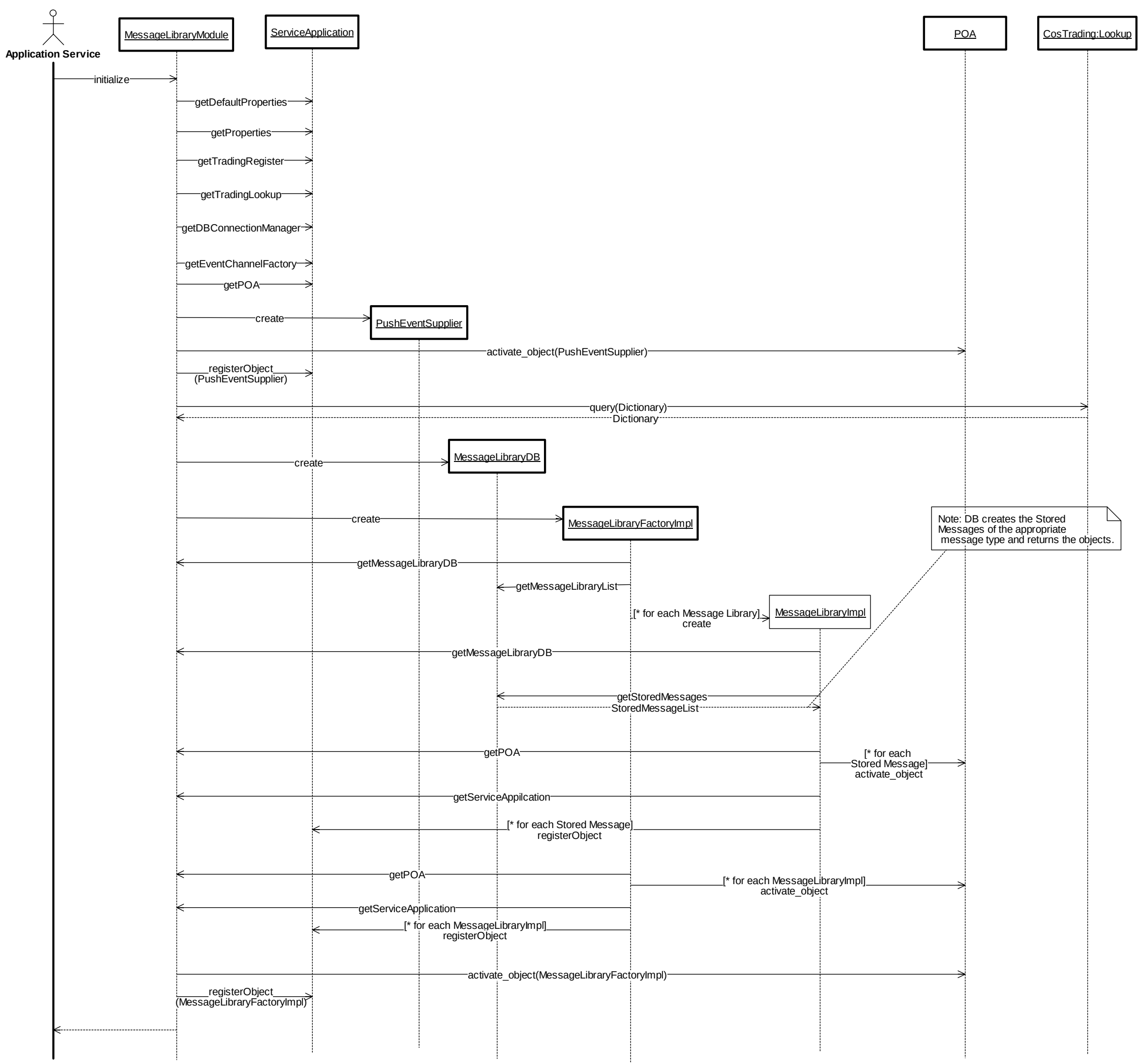

Figure 104. MessageLibraryModule:Initialize (Sequence Diagram)

3.11.2.7 MessageLibraryModule:IsMessageLibraryUsedByAnyPlan(Sequence Diagram)

This sequence diagram shows how a user can check if a plan is using the stored messages of a particular message library.

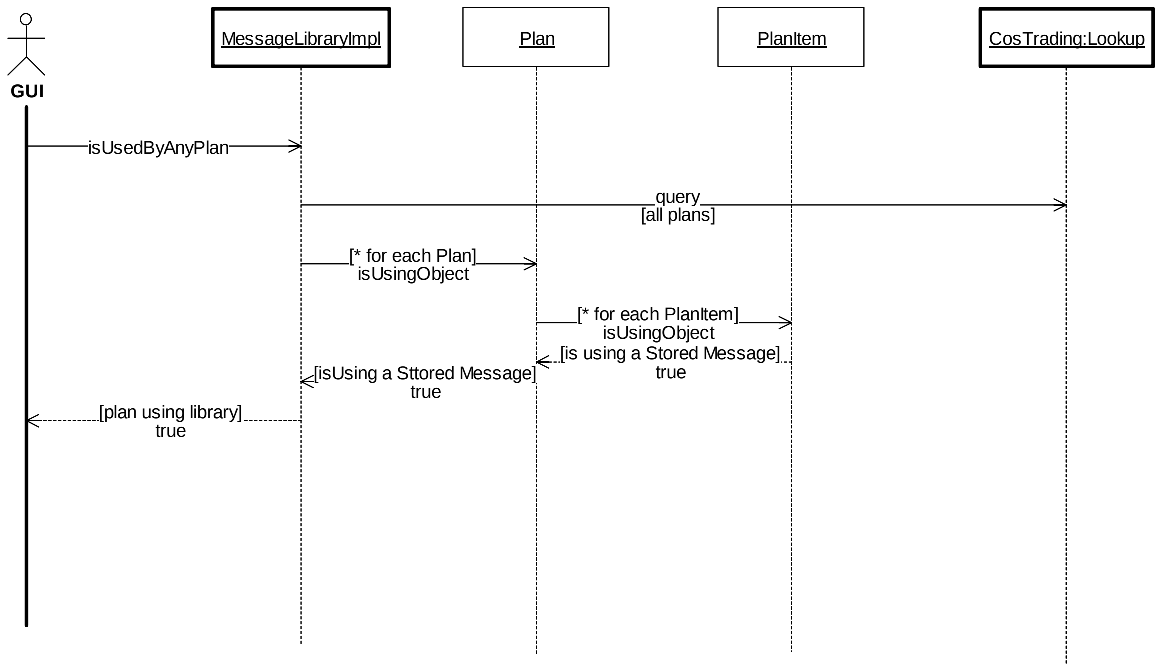

Figure 105. MessageLibraryModule:IsMessageLibraryUsedByAnyPlan (Sequence Diagram)

3.11.2.8 MessageLibraryModule:IsStoredMessageUsedByAnyPlan (Sequence Diagram)

This sequence diagram shows how a user can check if a plan is using a particular stored message.

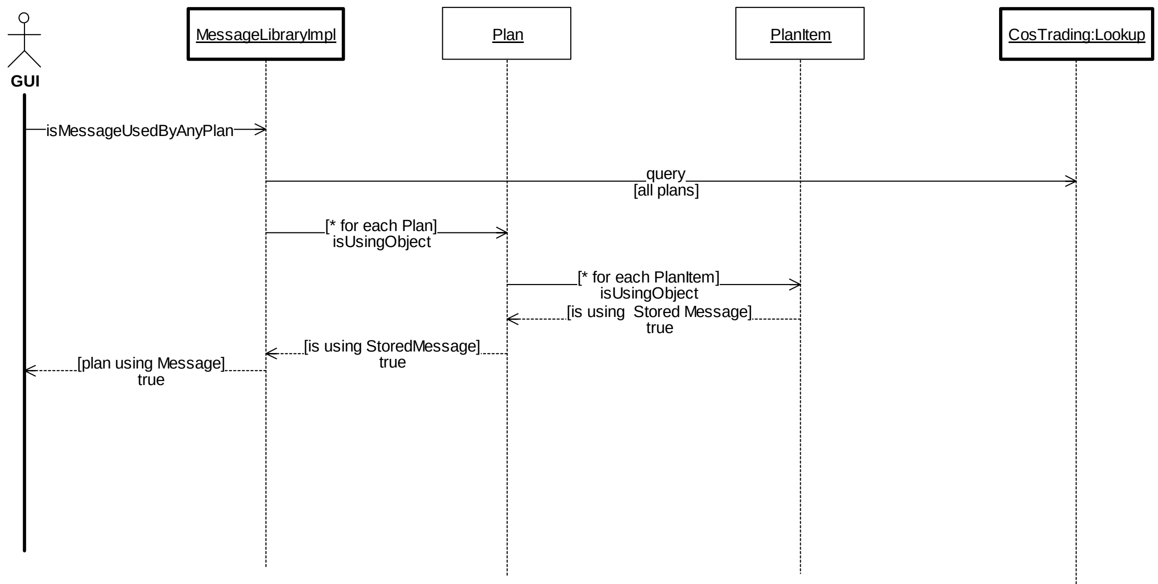

Figure 106. MessageLibraryModule:IsStoredMessageUsedByAnyPlan (Sequence Diagram)

3.11.2.9 MessageLibraryModule:ModifyDMSStoredMessage (Sequence Diagram)

A user with the proper functional rights can edit a stored message. The proposed contents for the stored message are checked against the dictionary prior to allowing the new content to be set. The state of the beacons associated with the message is also checked to make sure the beacons are not turned on for a message with no text. An event is pushed via the CORBA Event Service to notify others of the change to the stored message's contents. The user and operation details are logged in the operations log.

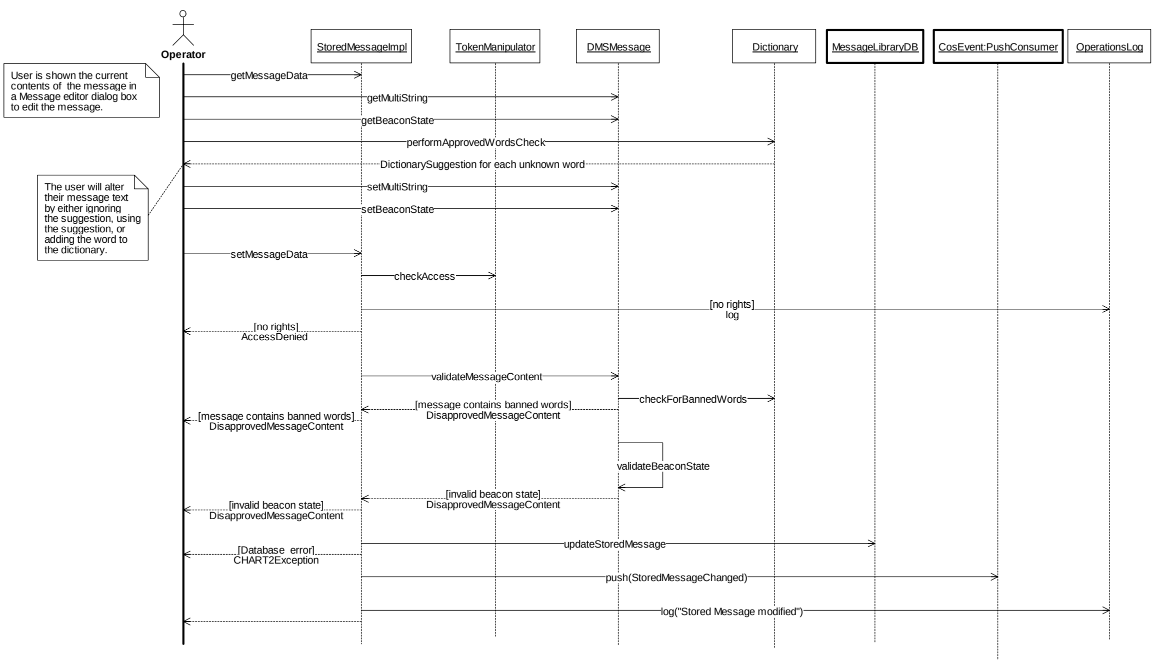

Figure 107. MessageLibraryModule:ModifyDMSStoredMessage (Sequence Diagram)

3.11.2.10 MessageLibraryModule:ModifyHARStoredMessage (Sequence Diagram)

A user with the proper functional rights can edit a stored HAR message. The proposed contents for the stored message are checked against the dictionary if it is in text format. An event is pushed via the CORBA Event Service to notify others of the change to the stored message's contents. The user and operation details are logged in the operations log.

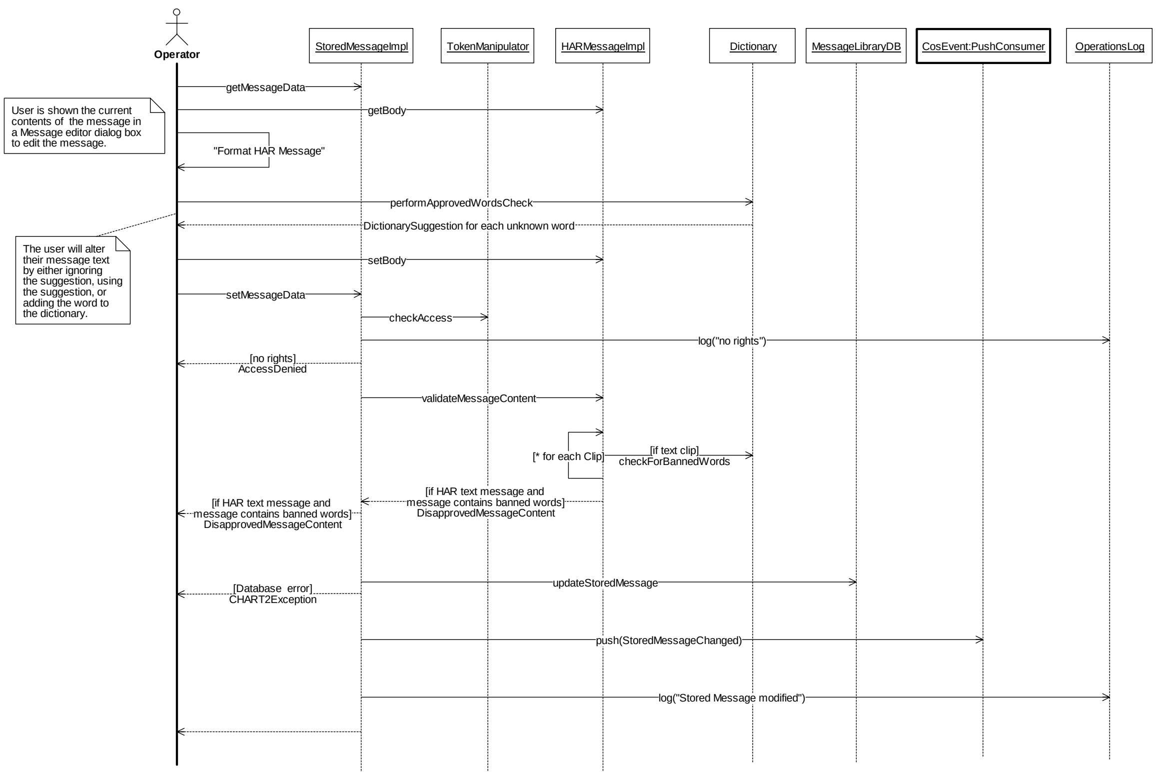

Figure 108. MessageLibraryModule:ModifyHARStoredMessage (Sequence Diagram)

3.11.2.11 MessageLibraryModule:SetLibraryName (Sequence Diagram)

A user with the proper functional rights may set the name assigned to a message library. An event is pushed via the CORBA Event Service to notify others of the name change. The user and operation details are logged in the operations log.

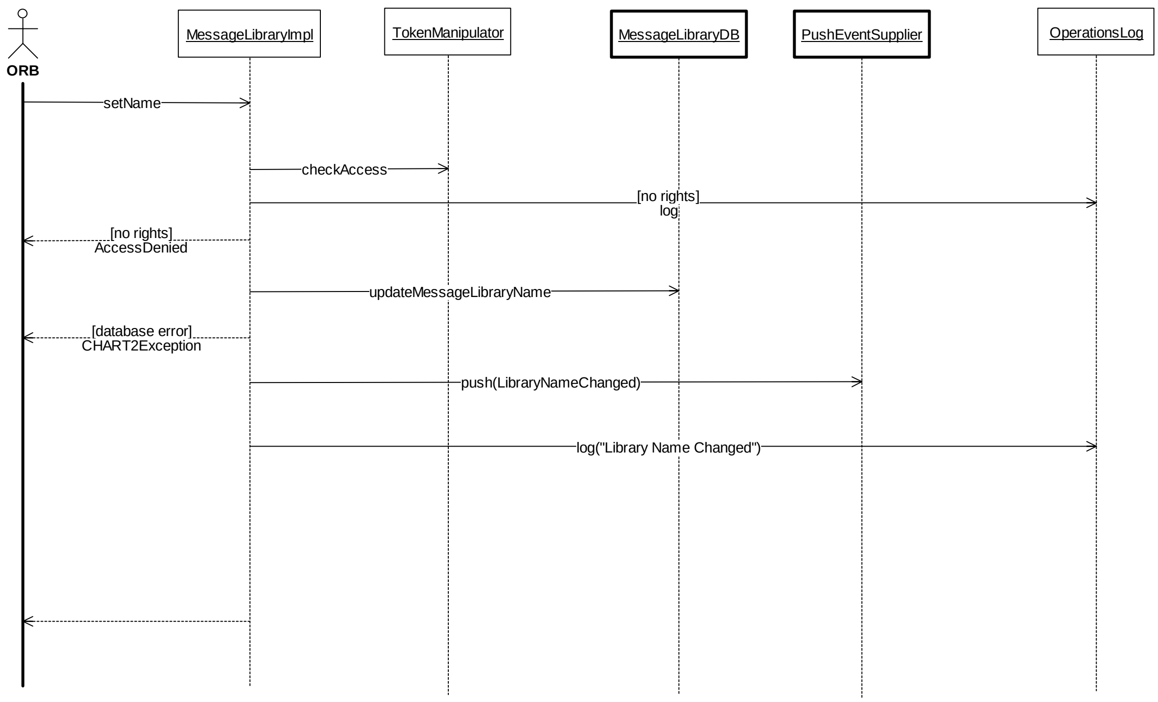

Figure 109. MessageLibraryModule:SetLibraryName (Sequence Diagram)

3.11.2.12 MessageLibraryModule:Shutdown (Sequence Diagram)

The MessageLibraryModule is shutdown by its host application. When told to shutdown, the MessageLibraryModule deactivates the MessageLibraryFactory from the POA, and shuts down the object. When the MessageLibraryFactory is shut down, deactivates each library from the POA and shuts down the object. The MessageLibrary deactivates any StoredMessage objects that it is serving.

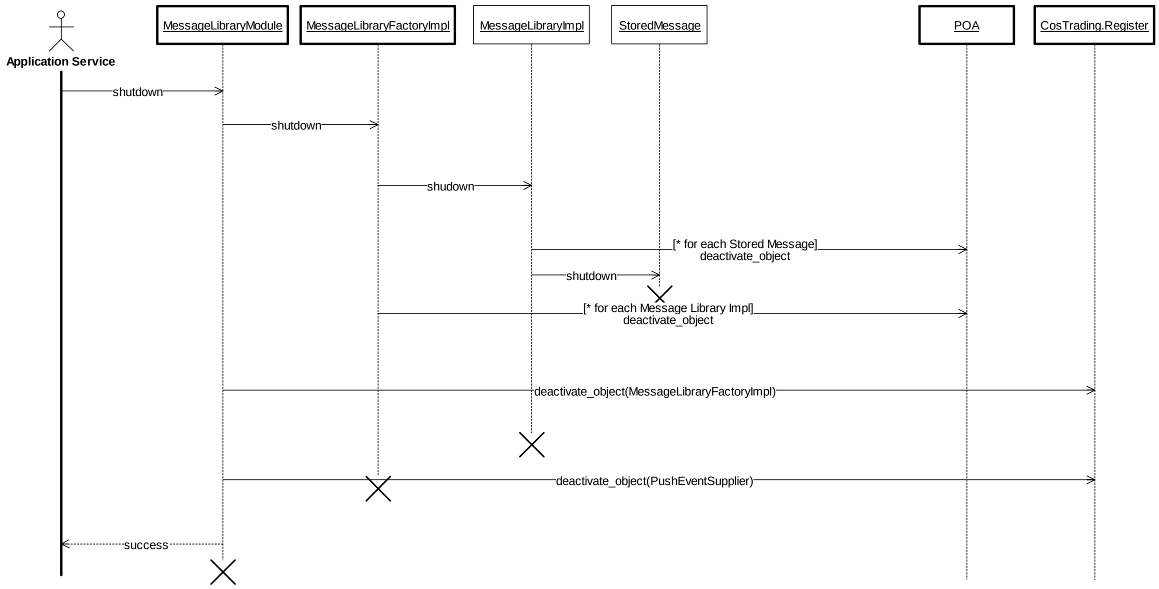

Figure 110. MessageLibraryModule:Shutdown (Sequence Diagram)

3.11.2.13 MessageLibraryModule:ViewDMSStoredMessage (Sequence Diagram)

The GUI discovers the contents of a DMS stored message during startup. The GUI is notified of changes to the contents of the DMS stored message via a CORBA event channel. When notified of such changes, the GUI updates itself so the user is always shown the latest information pertaining to the DMS stored message. The user and operation details are logged in the operations log.

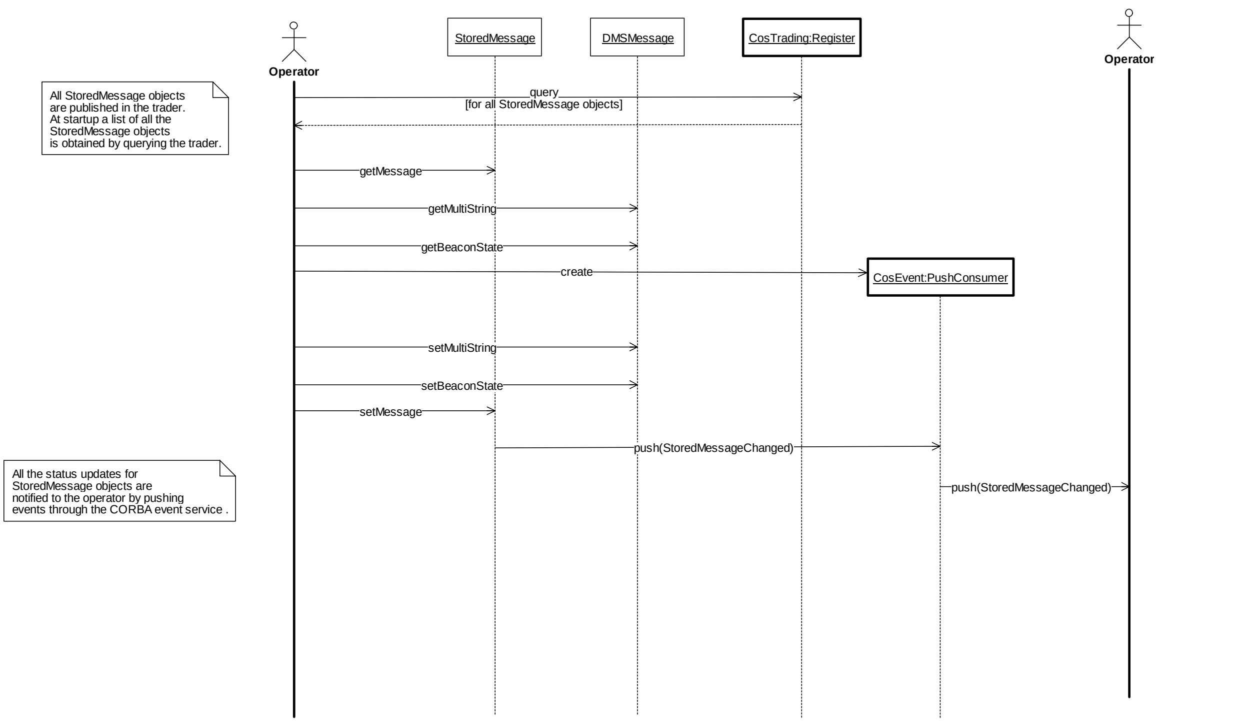

Figure 111. MessageLibraryModule:ViewDMSStoredMessage (Sequence Diagram)

3.11.2.14 MessageLibraryModule:ViewHARStoredMessage (Sequence Diagram)

The GUI discovers the contents of a HAR stored message during startup. The GUI is notified of changes to the contents of the HAR stored message via a CORBA event channel. When notified of such changes, the GUI updates itself so the user is always shown the latest information pertaining to the HAR stored message. The user and operation details are logged in the operations log.

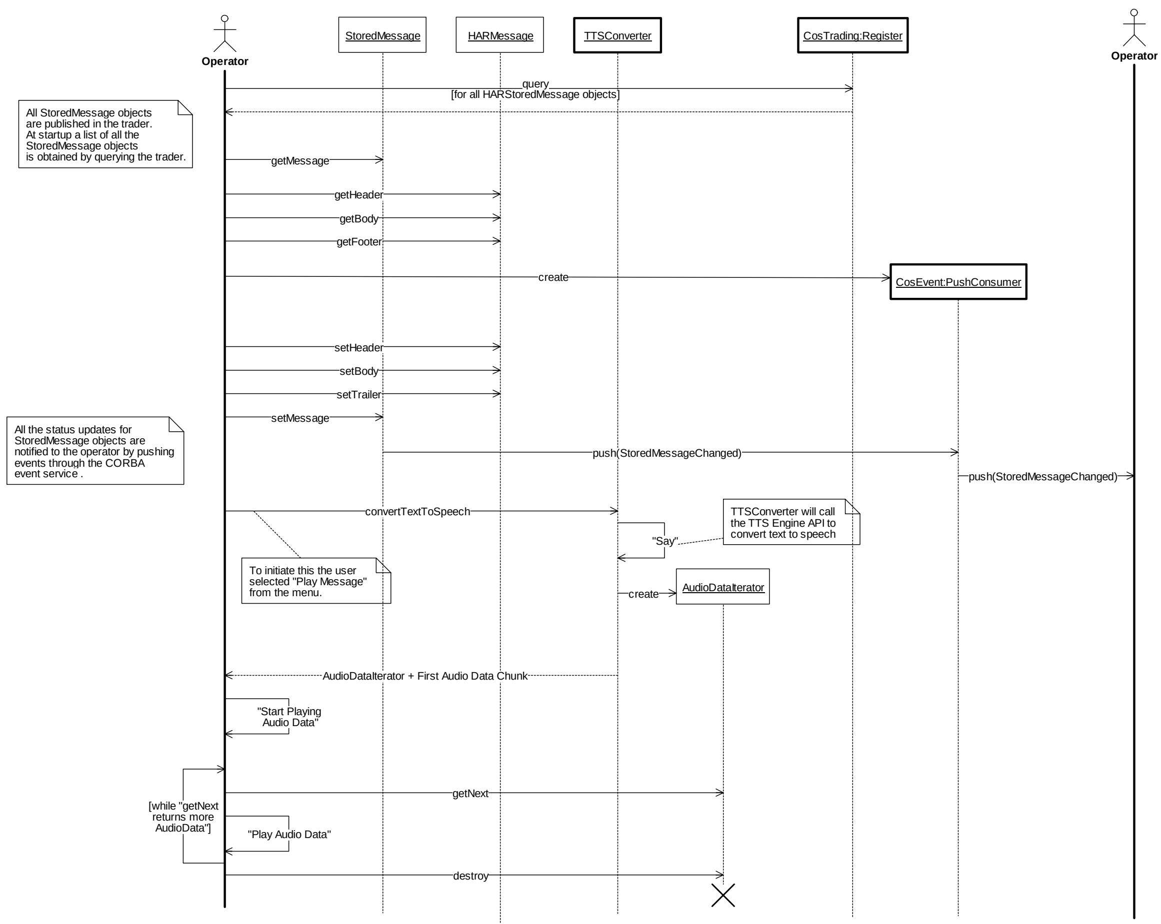

Figure 112. MessageLibraryModule:ViewHARStoredMessage (Sequence Diagram)

3.12 PlanModule

3.12.1 Classes

3.12.1.1 PlanModuleClasses (Class Diagram)

This is an installable module that serves the PlanFactory, Plan, and PlanItem objects to the rest of the CHART2 system.

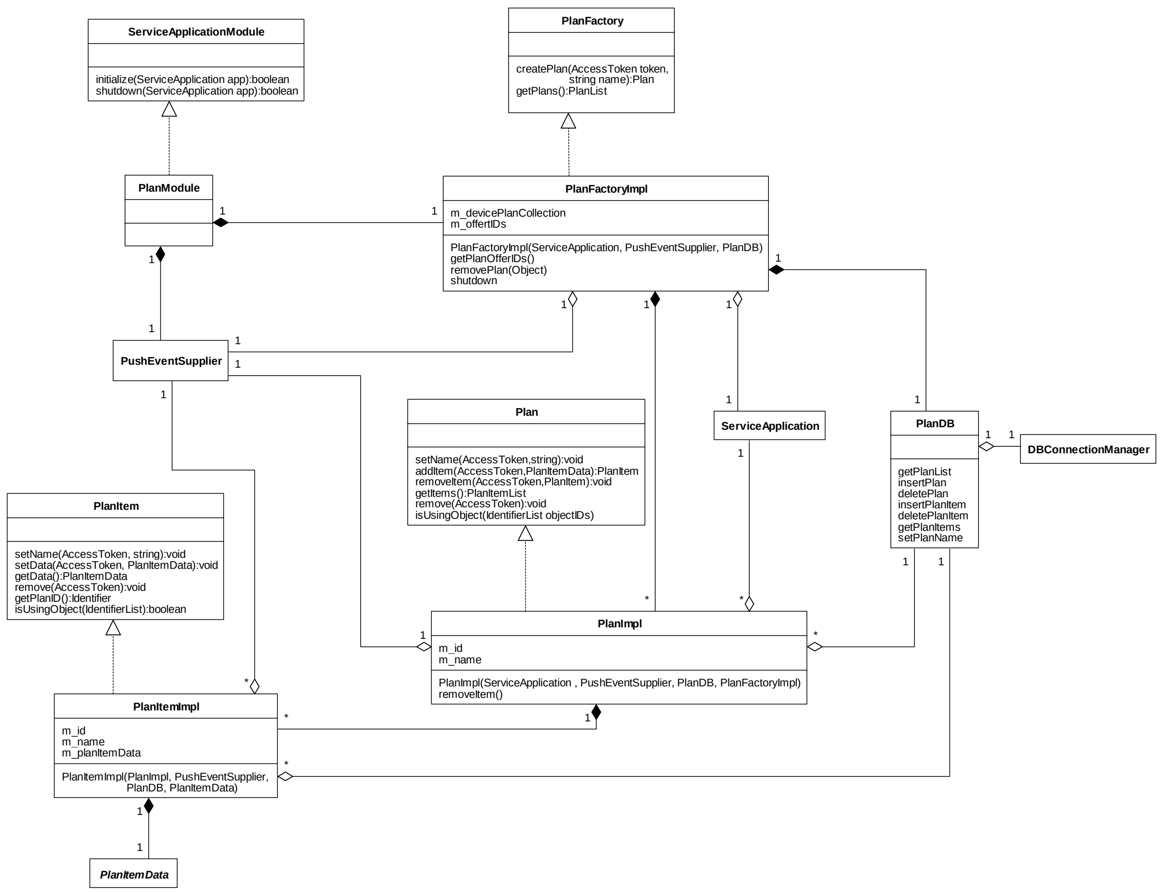

Figure 113. PlanModuleClasses (Class Diagram)

3.12.1.1.1 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two separate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the “jdbc.drivers” system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.12.1.1.2 Plan (Class)

A Plan is a group of actions listed out in advance to be used in response to a traffic event. Each action is defined to be a Plan item. The Plan supports functionality to add and remove plan items.

3.12.1.1.3 PlanDB (Class)

This class contains the methods that perform database operations for the Plan module. It is constructed with a Database object that provides the connections to the database server. All the methods in this class get a new connection to the database before performing any operation on the database. The connection is released at completion of the operation.

3.12.1.1.4 PlanFactory (Class)

This class creates, destroys, and maintains the collection of plans that can be used in the system.

3.12.1.1.5 PlanFactoryImpl (Class)

This class implements the PlanFactory interface and enables the management of the Plan objects by other processes. It creates, publishes and deletes the objects that implement the Plan interface.

3.12.1.1.6 PlanImpl (Class)

This class implements the Plan interface and provides the implementation for the methods defined in the interface. It also manages the database operations for the PlanItems contained in this Plan.

3.12.1.1.7 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This CORBA interface is subclassed for specific actions that can be planned in the system.

3.12.1.1.8 PlanItemData (Class)

This class is a valuetype that is the base class for data stored in a plan item. Derived classes contain specific data that map a device to an operation and the data needed for the operation. For example a derived class provides a mapping between a specific DMS and a DMSMessage.

3.12.1.1.9 PlanItemImpl (Class)

This class implements the PlanItem interface.

3.12.1.1.10 PlanModule (Class)

This module creates, publishes and deletes the objects that implement the PlanFactory interface.

3.12.1.1.11 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier’s push rate.

3.12.1.1.12 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.12.1.1.13 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.12.2 Sequence Diagrams

3.12.2.1 PlanModule:AddItem (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can add an item to an existing plan in the system. An AccessDenied exception is returned if the user does not have the right to add an item to the plan. Otherwise, a PlanItem object is created and added to the database. A PlanItemAdded event is pushed through the event channel to notify other processes that a plan item has been added to this plan. User actions are logged to the operations log.

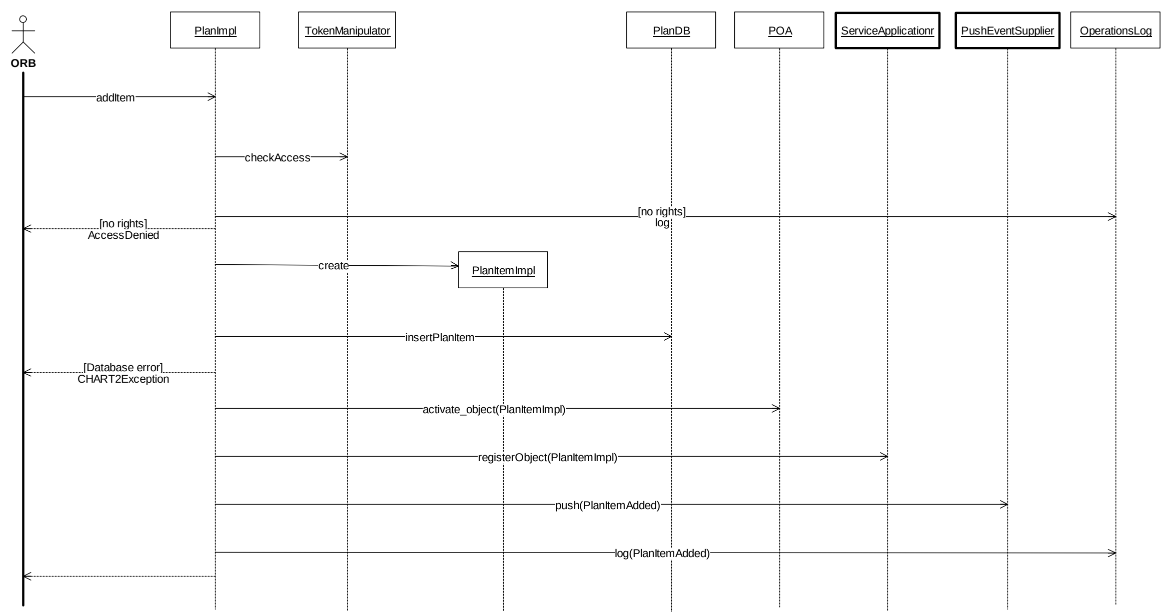

Figure 114. PlanModule:AddItem (Sequence Diagram)

3.12.2.2 PlanModule:AddPlan (Sequence Diagram)

This diagram shows how a user with proper functional rights can add a plan to the system. An AccessDenied exception is returned if the user does not have the functional right to add a plan. Otherwise, the plan object is created and added to the database. The plan object is published in CORBA Trader service and a PlanAdded event is pushed through the event channel to notify the other processes that a new plan has been added.

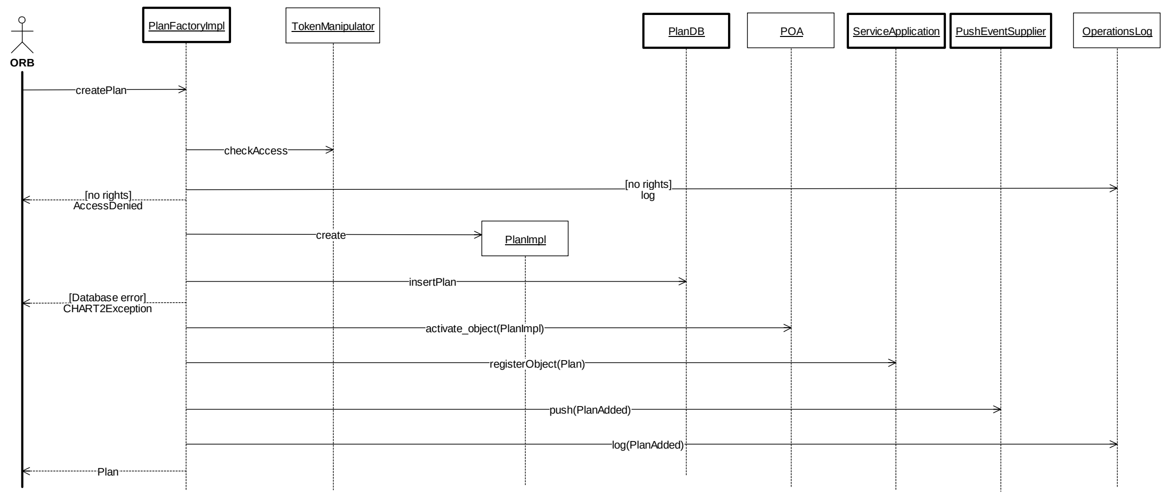

Figure 115. PlanModule:AddPlan (Sequence Diagram)

3.12.2.3 PlanModule:Initialize (Sequence Diagram)

This sequence diagram shows the startup for the Plan Module. An ApplicationService will initialize this module. The references to basic services such as POA, Trader, Event channel and database are obtained from the ServiceApplication. This module creates a Plan Module specific database object. It also creates the PlanFactory object, which creates the Plan objects from the plan list obtained from the database. The Plan objects are published in the trader. An event channel is created to push the events to clients and it is published in the trader register. The Offer IDs of all the objects that were published in the trader are saved to a file so that they may be withdrawn.

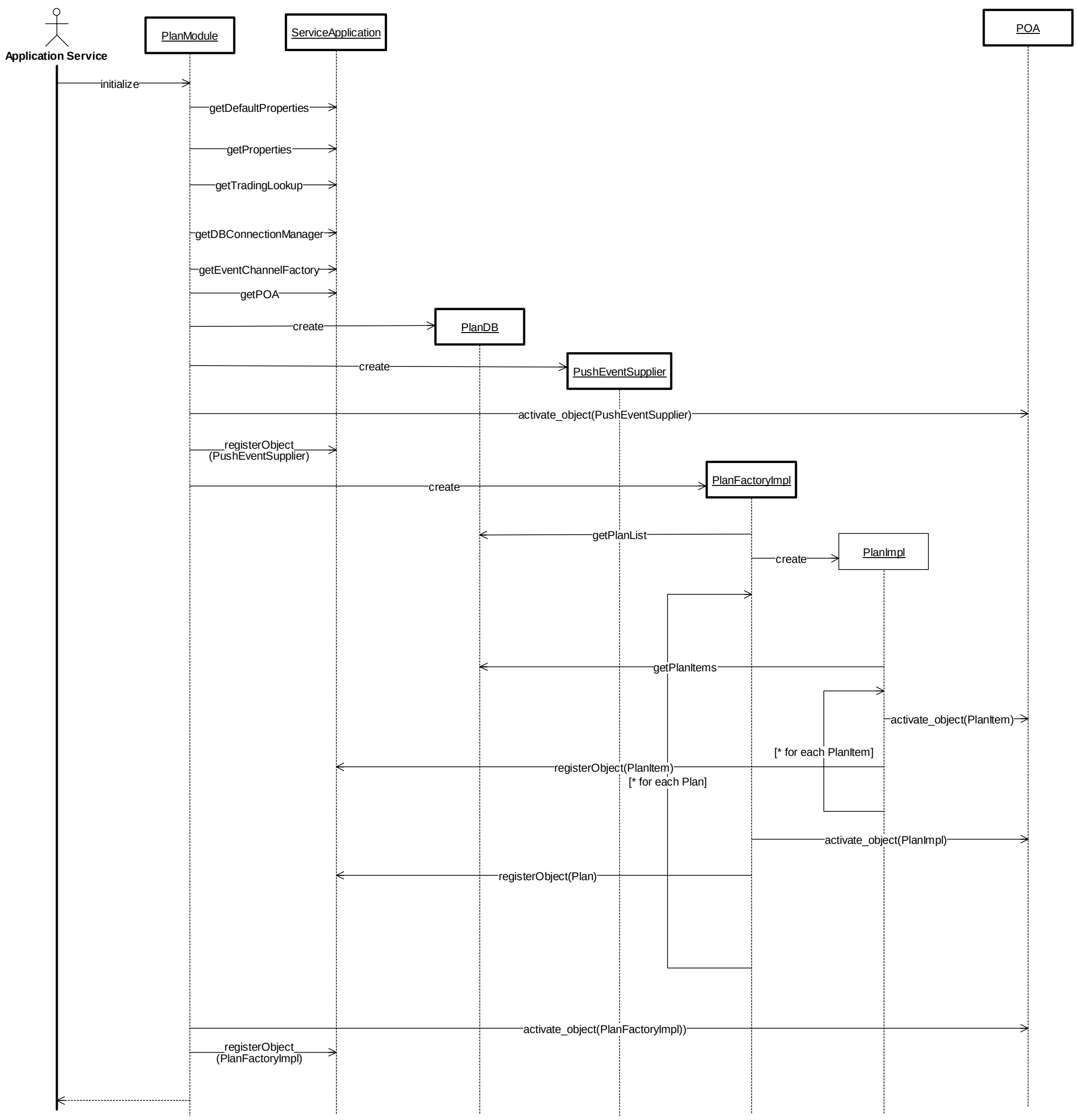

Figure 116. PlanModule:Initialize (Sequence Diagram)

3.12.2.4 PlanModule:PlanIsUsingObject (Sequence Diagram)

This sequence diagrams shows how to check if a plan is using a particular set of objects. The IDs of the object are passed to the Plan object to check if its PlanItems are using these objects. If a PlanItem is using any object, the Plan returns true.

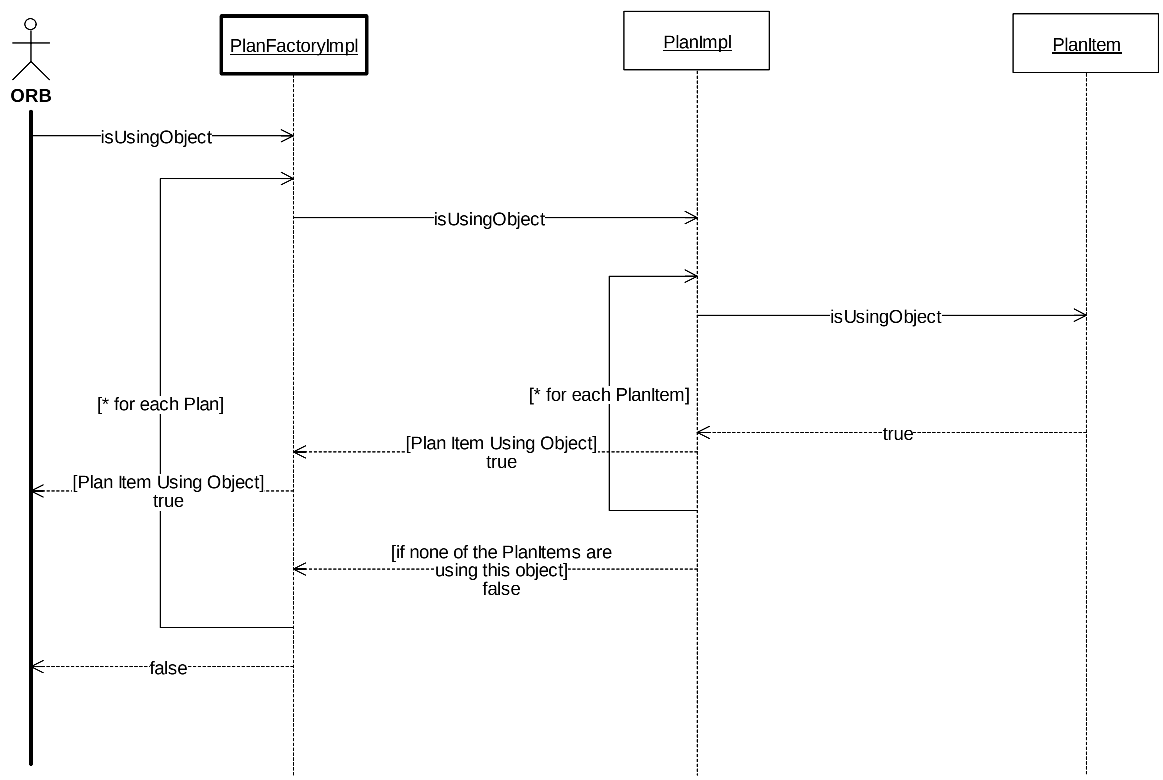

Figure 117. PlanModule:PlanIsUsingObject (Sequence Diagram)

3.12.2.5 PlanModule:PlanItemIsUsingObject (Sequence Diagram)

This sequence diagrams shows how to check if a plan item is using an object from a set of objects. The IDs of the objects are passed to the PlanItem object. If the PlanItem is using any object, it returns true.

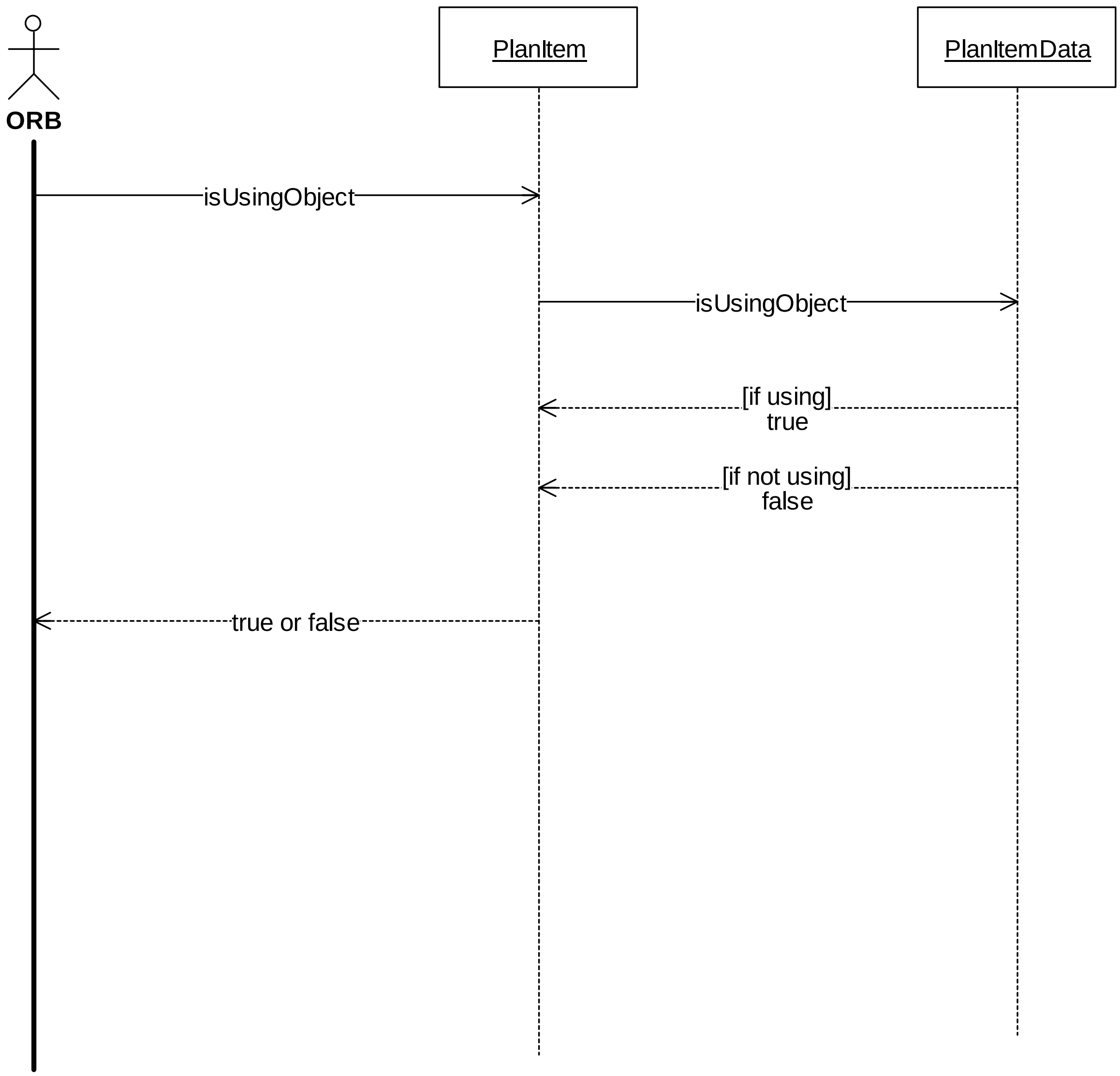

Figure 118. PlanModule:PlanItemIsUsingObject (Sequence Diagram)

3.12.2.6 PlanModule:RemoveItem (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can remove a plan item from a plan in the system. An AccessDenied exception is returned if the user does not have the right to remove an item from the plan. Otherwise, the plan item is deleted from the database and the object is destroyed. An event is pushed through the event channel to notify other processes that the plan item has been removed from the plan. User actions are logged to the operations log.

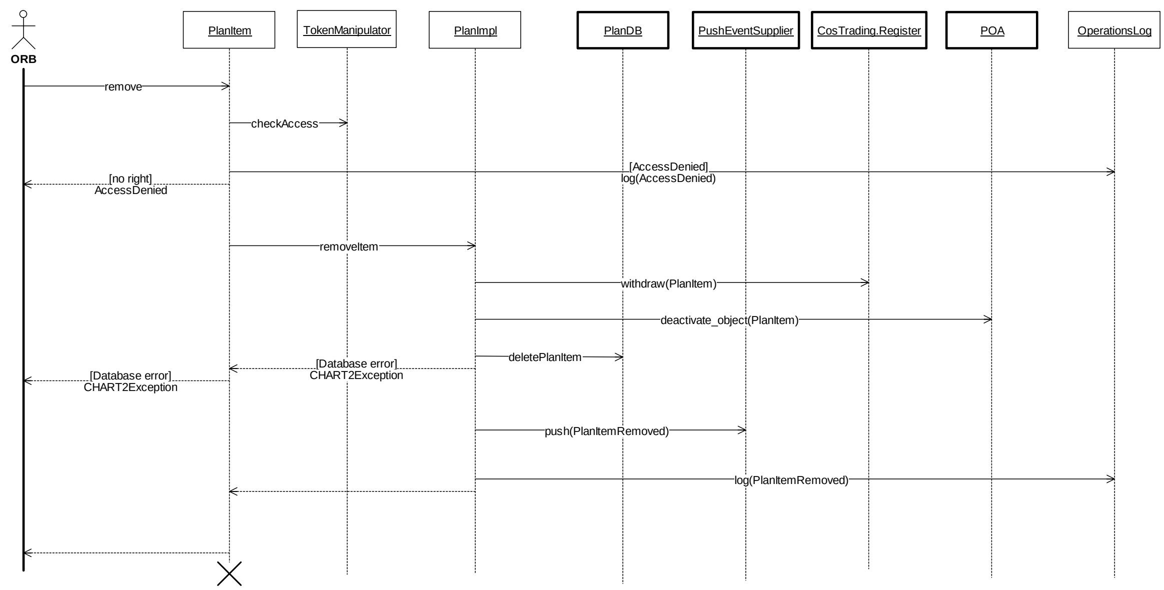

Figure 119. PlanModule:RemoveItem (Sequence Diagram)

3.12.2.7 PlanModule:RemovePlan (Sequence Diagram)

This sequence diagram shows how a user with proper rights can delete a Plan from the system. An AccessDenied exception is returned if the user does not have the functional right to delete a Plan. Otherwise, the Plan is deleted from the database and the object is destroyed. The Plan is withdrawn from the trader and a PlanRemoved event is pushed through the event channel to notify the clients that the plan has been deleted. Note that the deletion of a plan results in the deletion of all the plan items that are used in the plan from the system and the database. The user actions are logged to the operations log.

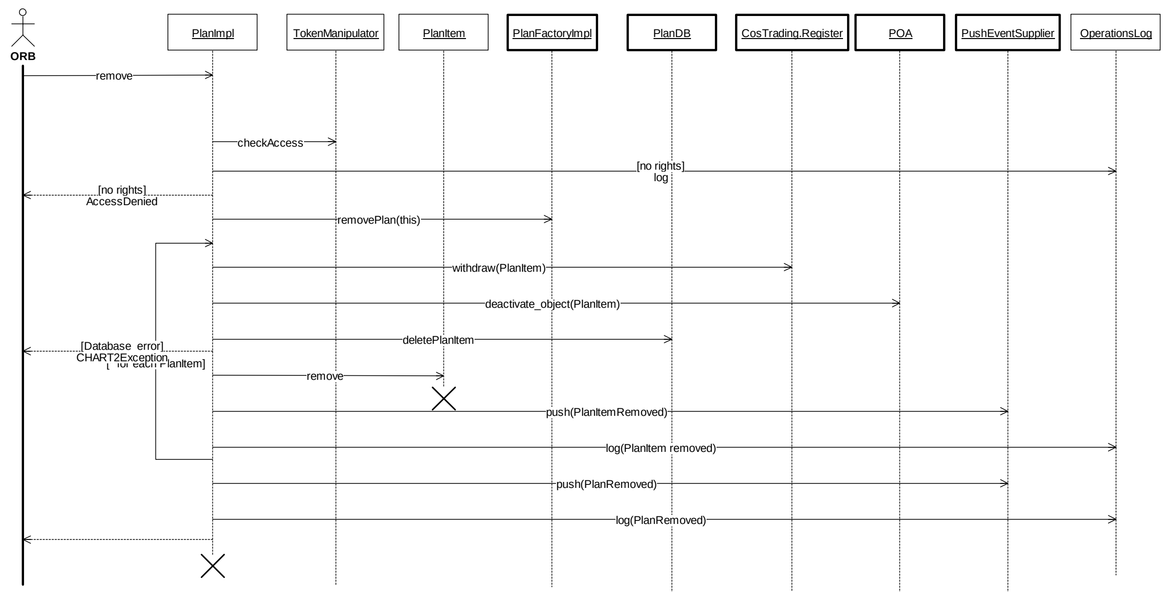

Figure 120. PlanModule:RemovePlan (Sequence Diagram)

3.12.2.8 PlanModule:RemovePlanFromFactory (Sequence Diagram)

This sequence diagram shows how a Plan object is removed from the Plan Factory when a Plan is deleted from the system.

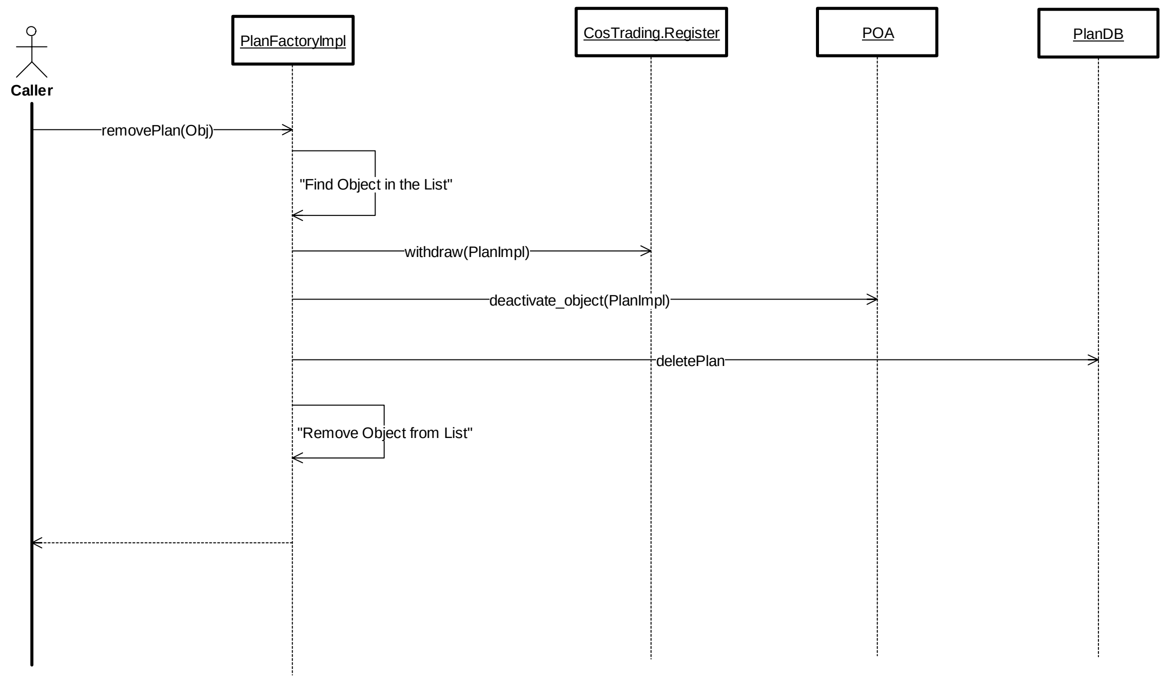

Figure 121. PlanModule:RemovePlanFromFactory (Sequence Diagram)

3.12.2.9 PlanModule:SetPlanItemData (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can change the PlanItemData object of a plan item. An AccessDenied exception is returned if the user does not have the right to modify the plan item. Otherwise, the PlanItemData is updated and stored in the database. An event is pushed through the event channel to notify other processes that the plan item has been changed. User actions are logged to the operations log.

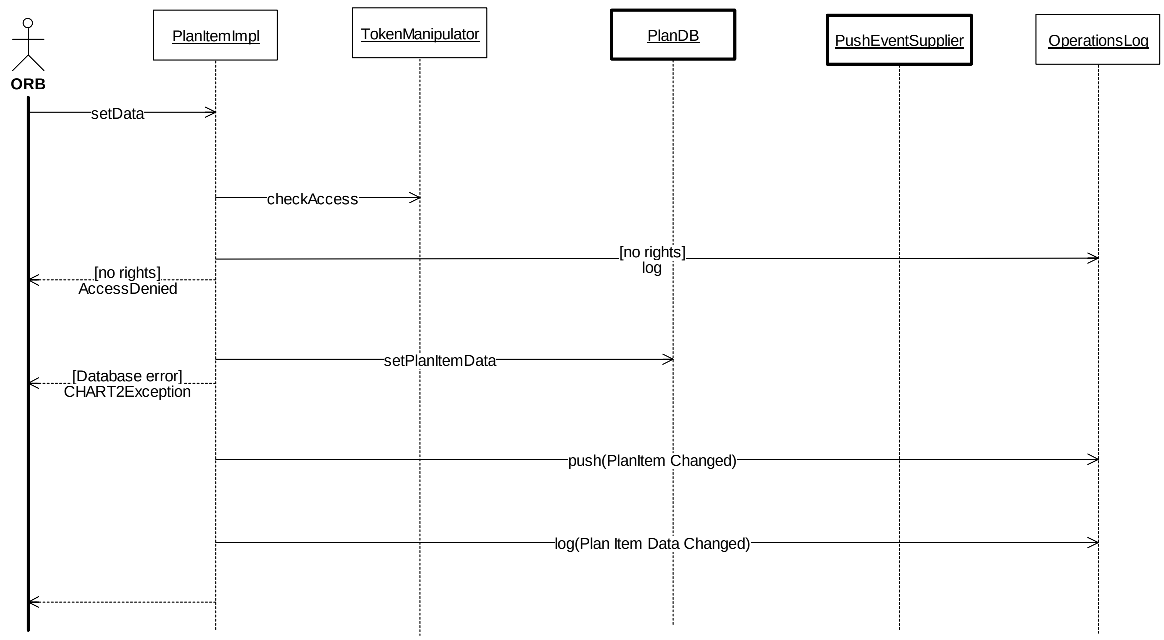

Figure 122. PlanModule:SetPlanItemData (Sequence Diagram)

3.12.2.10 PlanModule:SetPlanItemName (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can change the name of a plan item. An AccessDenied exception is returned if the user does not have the right to change the plan item name. Otherwise, the plan item name is changed and stored in the database. An event is pushed through the event channel to notify other processes that the plan item has been changed. User actions are logged to the operations log.

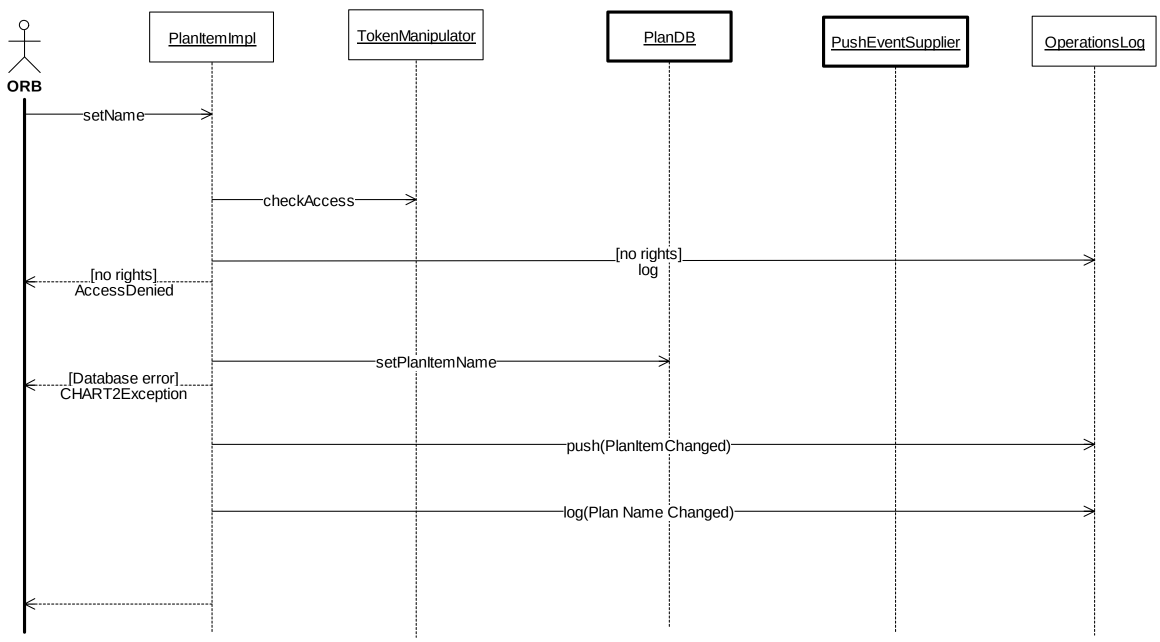

Figure 123. PlanModule:SetPlanItemName (Sequence Diagram)

3.12.2.11 PlanModule:SetPlanName (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can set the name of a Plan. An access denied exception is returned if the user does not have the right to change the name. Otherwise, the name is changed and the database is updated. An event id pushed via the CORBA event service to notify others of the new Plan name. The user actions are logged to the operations log.

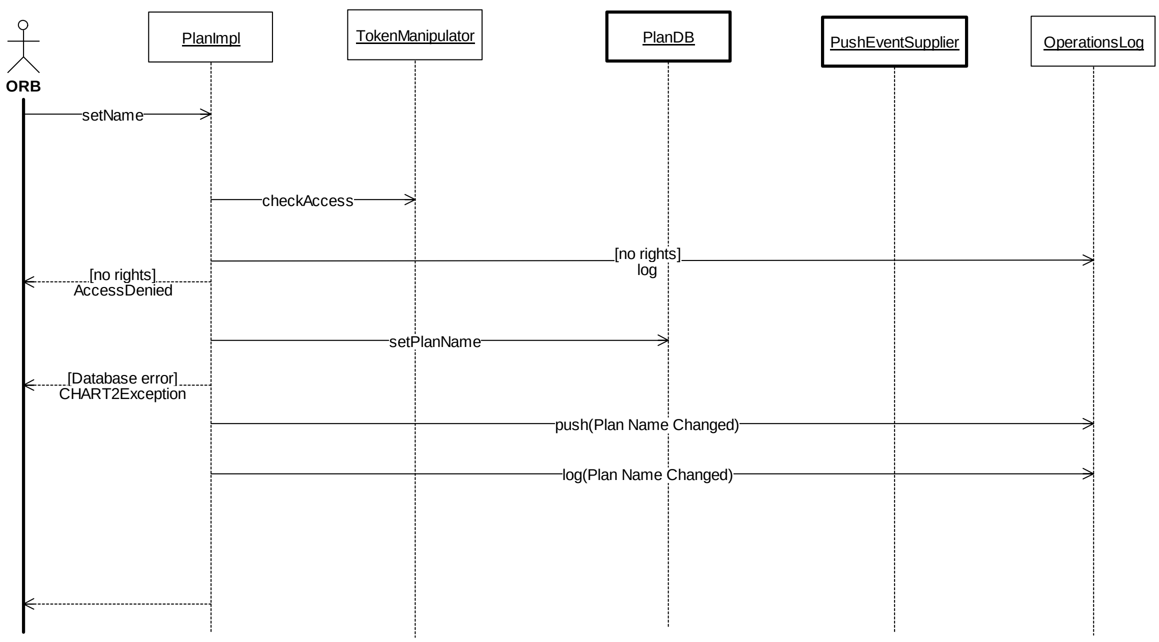

Figure 124. PlanModule:SetPlanName (Sequence Diagram)

3.12.2.12 PlanModule:Shutdown (Sequence Diagram)

This diagram shows the shutdown sequence of the Plan module. All the Plan objects that were published in the trader by the PlanFactory and the PlanFactory itself are withdrawn and destroyed. The event channel is also withdrawn from the trader and destroyed.

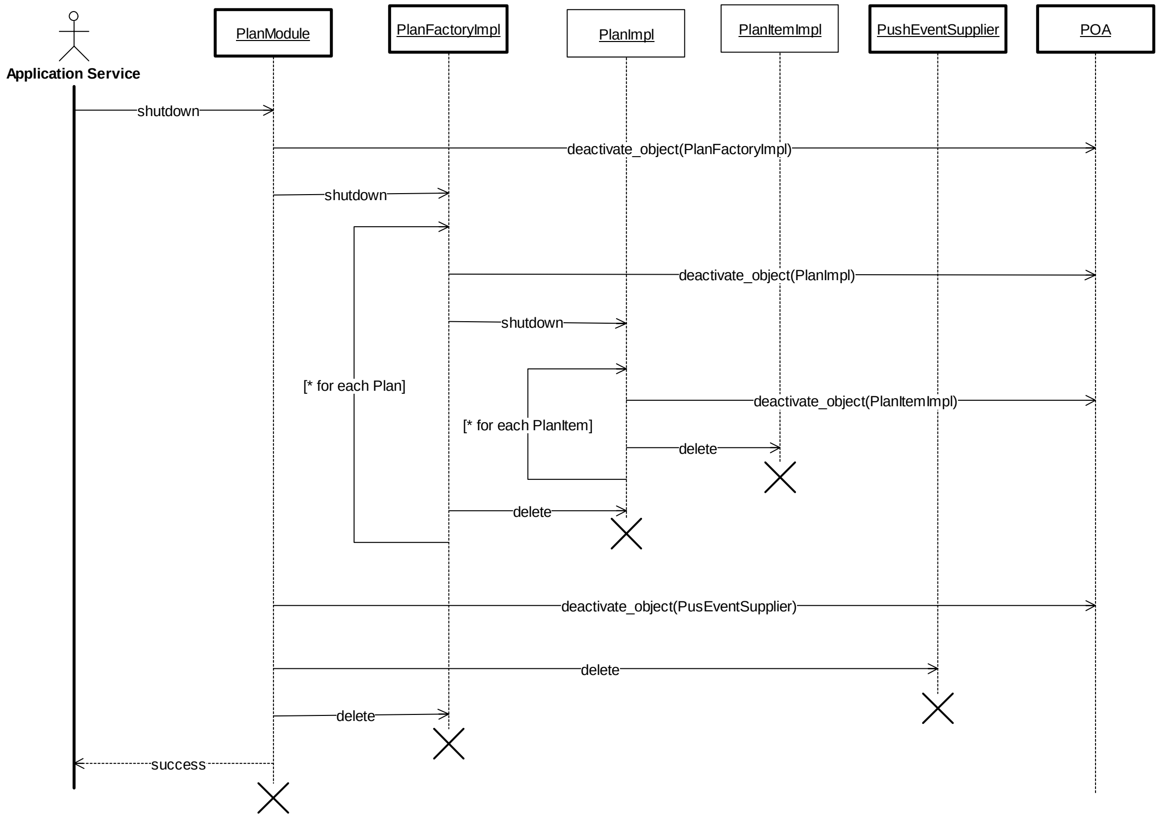

Figure 125. PlanModule:Shutdown (Sequence Diagram)

3.13 ResourcesModule

3.13.1 Classes

3.13.1.1 ResourceClasses (Class Diagram)

This diagram shows the classes in the ResourcesModule, an installable service module that serves objects that implement the Organization and OperationsCenter interfaces.

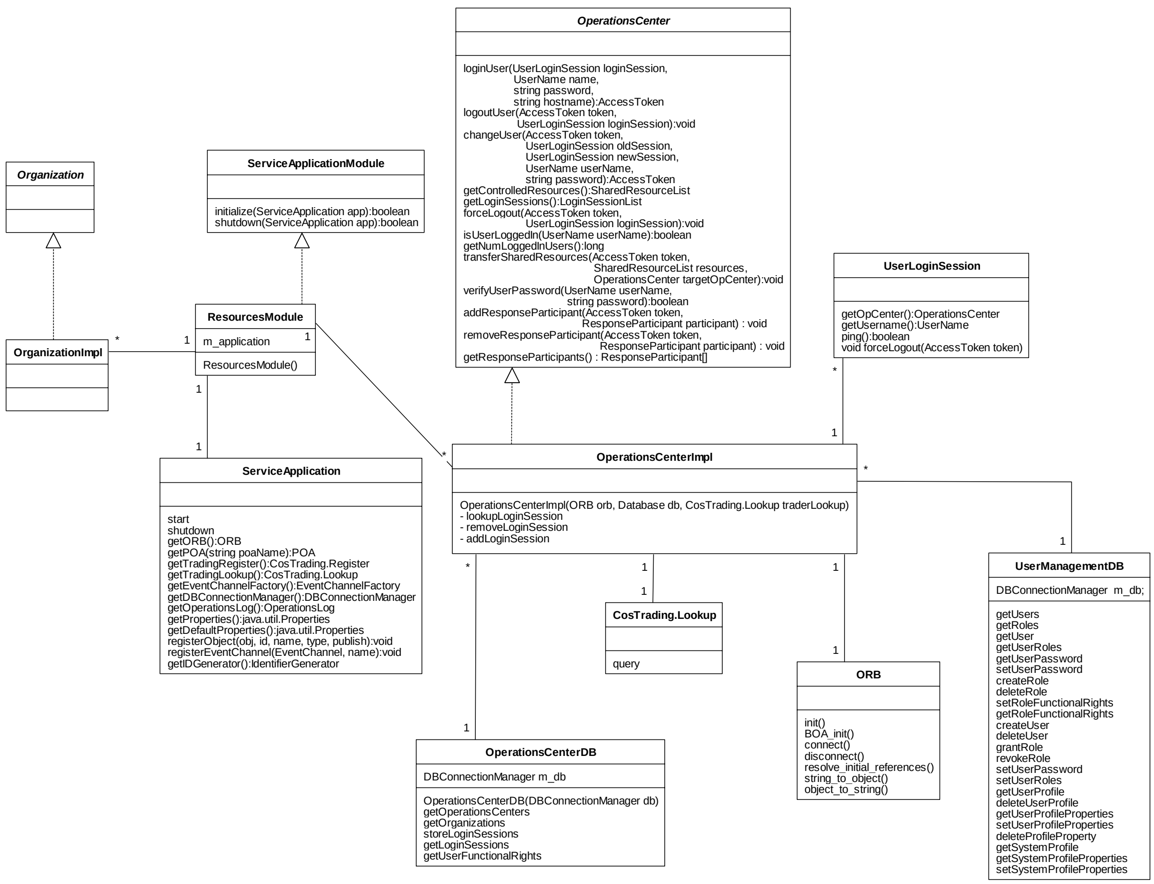

Figure 126. ResourceClasses (Class Diagram)

3.13.1.1.1 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects that have previously been published.

3.13.1.1.2 OperationsCenter (Class)

The OperationsCenter represents a center where one or more users are located. This class is used to log users into the system. If the username and password provided to the loginUser method are valid, the caller is given a token that contains information about the user and the functional rights of the user. This token is then used to call privileged methods within the system. Shared resources in the system are either available or under the control of an OperationsCenter. The OperationsCenter keeps track of users that are logged in so that it can ensure that the last user does not log out while there are shared resources under its control. This list of logged in users is also available for monitoring system usage or to force users to logout for system maintenance.

3.13.1.1.3 OperationsCenterDB (Class)

This class provides a set of API calls to access the Operations Center data from the database. The API's provide functionality to add, remove and retrieve Operation Center data from the database. The connection to the database is acquired from the Database object that manages all the database connections.

3.13.1.1.4 OperationsCenterImpl (Class)

This class provides the implementation of the OperationsCenter interface for this module. It, therefore, provides a concrete implementation of each of the methods in the interface. It also contains a collection of UserLoginSession objects, one for each user who is currently logged in.

3.13.1.1.5 ORB (Class)

The CORBA ORB (Object Request Broker) provides a common object oriented, remote procedure call mechanism for inter-process communication. The ORB is the basic mechanism by which client applications send requests to server applications and receive responses to those requests from servers.

3.13.1.1.6 Organization (Class)

The Organization interface extends the UniquelyIdentifiable interface and will represent an organization, that is an administrative body that can control or own resources.

3.13.1.1.7 OrganizationImpl (Class)

This class provides the implementation of the Organization interface for this module. Thus, it provides a concrete implementation of each of the methods in the interface.

3.13.1.1.8 ResourcesModule (Class)

This module creates, publishes and destroys all objects related to resource management that are used by the User Management service application.

3.13.1.1.9 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.13.1.1.10 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.13.1.1.11 UserLoginSession (Class)

The UserLoginSession CORBA interface is used to store information about a user that is logged into the system. This object is served from the GUI and provides a means for the servers to call back into the GUI process.

3.13.1.1.12 UserManagementDB (Class)

The UserManagementDB Class provides methods used to access and modify User Managment data in the database. This class uses a Database object to retrieve a connection to the database for its exclusive use during a method call.

3.13.2 Sequence Diagrams

3.13.2.1 ResourcesModule:ChangeUser (Sequence Diagram)

A client with the correct functional rights may select to relinquish his/her workstation to another operator. This typically will happen at shift change. This sequence logs the new operator in before logging the old operator out, thereby guaranteeing that the shared resources controlled by the operations center have a responsible operator during the transition. If this method throws any type of exception, the old user is still logged in and the new user is not. If this method returns a token, the old user is logged out and the new user is logged in.

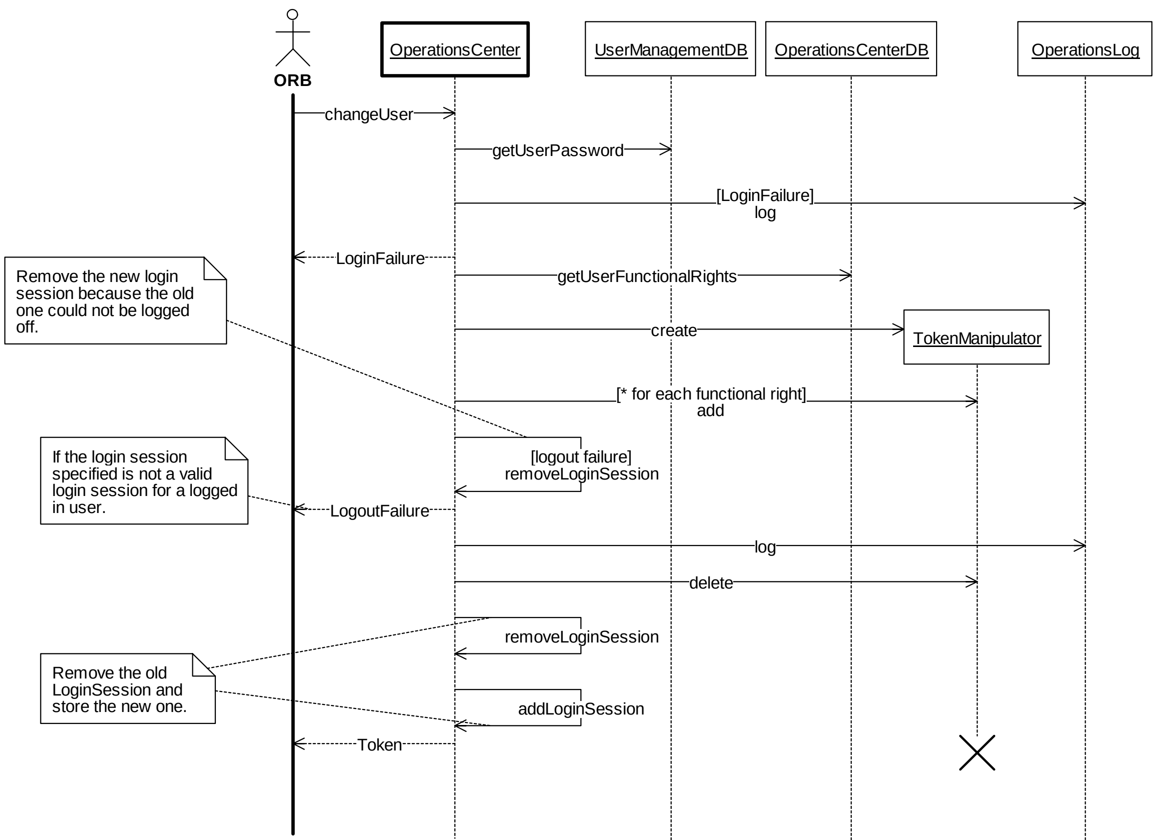

Figure 127. ResourcesModule:ChangeUser (Sequence Diagram)

3.13.2.2 ResourcesModule:ForceLogout (Sequence Diagram)

A client with the correct functional rights may force a particular user to logout of the CHART2 system. This is actually accomplished in two steps. The client would first need to acquire a UserLoginSession object before calling this method, please refer to the sequence diagram for the getUserLoginSessions method for details. Once the user has acquired a UserLoginSession he/she may contact the Operations Center where that UserLoginSession is being tracked and inform it that the user should be forced to logout. The OperationsCenter will call the forceLogout method on the specified UserLoginSession after removing the login session from its internal collection of login sessions. Note that it is possible for the user to call the forceLogout method directly on the UserLoginSession without informing the OperationsCenter. This method of forcing a user to logout is also accepted. If this path is taken, the operations center will contain a reference to a UserLoginSession that is no longer valid. This possibility is accounted for by pinging the UserLoginSession objects each time the getNumLoggedInUsers() method is called. Please refer to that sequence diagram for details.

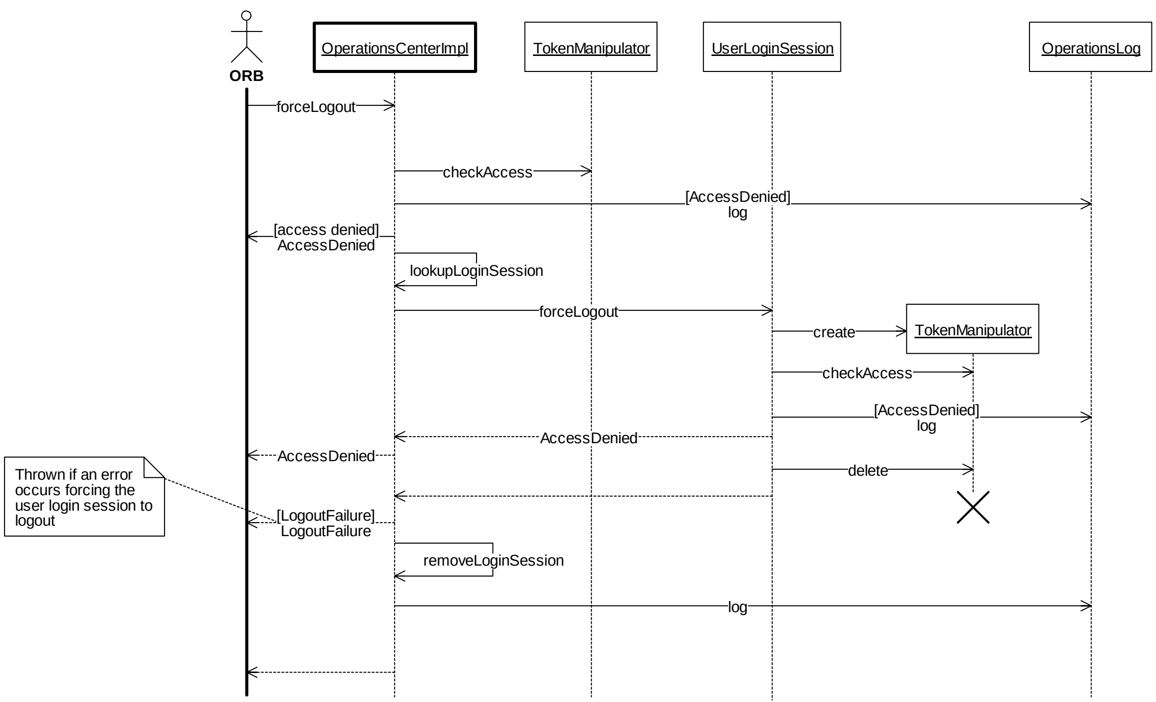

Figure 128. ResourcesModule:ForceLogout (Sequence Diagram)

3.13.2.3 ResourcesModule:GetControlledResources (Sequence Diagram)

A client may request a list of all shared resources that are currently controlled by this operations center. This would typically happen if the user were looking to transfer responsibility for some of all of the controlled shared resources from one operations center to another. The operations center will contact each shared resource manager in the system and get a list of resources that it is currently controlling. The lists returned by each shared resource manager will be combined and the entire list of controlled resources will be returned to the user.

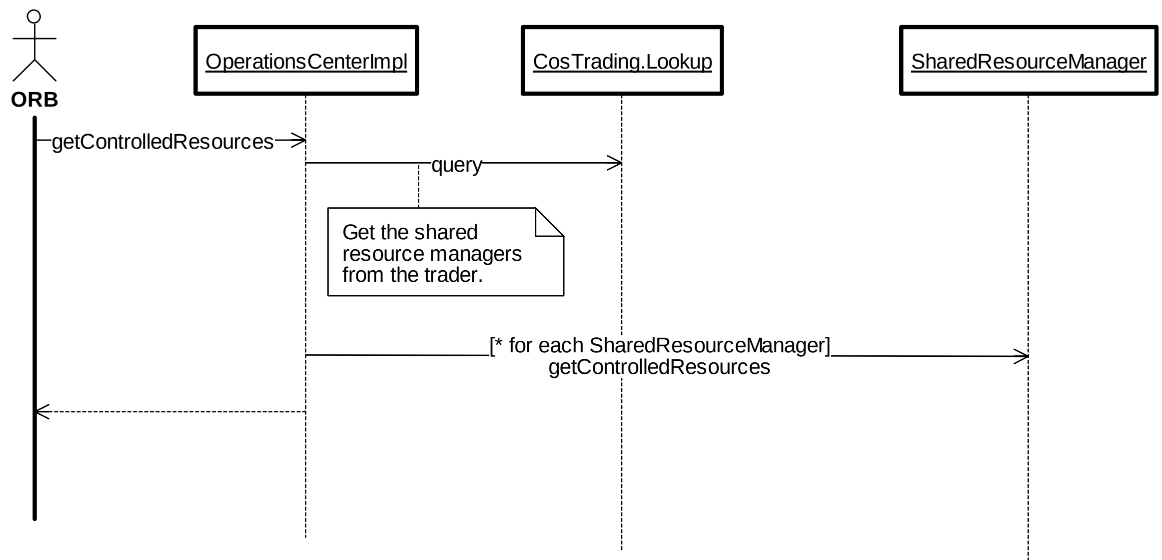

Figure 129. ResourcesModule:GetControlledResources (Sequence Diagram)

3.13.2.4 ResourcesModule:GetLoginSessions (Sequence Diagram)

A client with the correct functional rights may get a list of UserLoginSessions that represents the list of users who are currently logged in from this operations center.

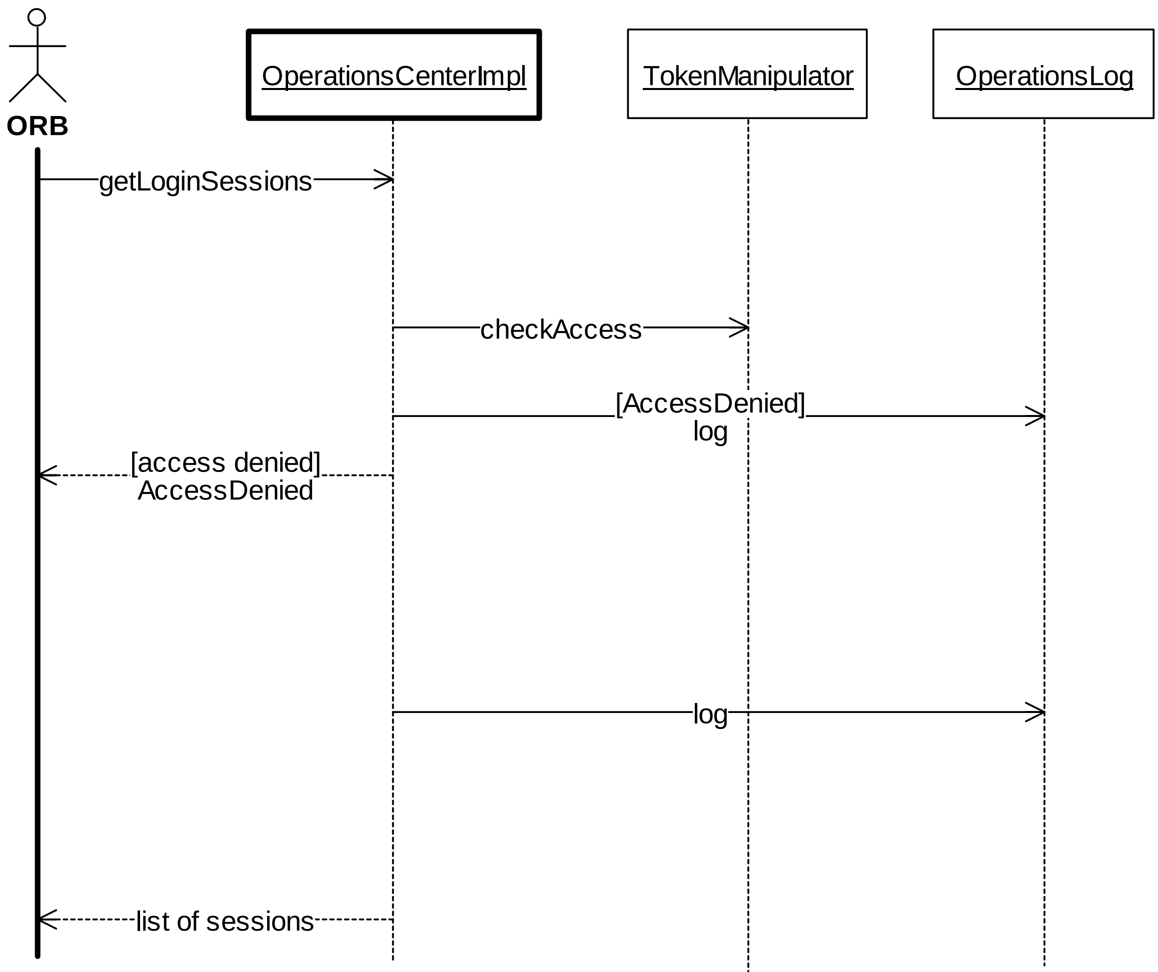

Figure 130. ResourcesModule:GetLoginSessions (Sequence Diagram)

3.13.2.5 ResourcesModule:GetNumLoggedInUsers (Sequence Diagram)

This method allows a client to get the number of users who are currently logged in at this operations center. This method will be used by the shared resource manager watchdogs to verify that they do not have shared resources which are under the control of operations centers with no users logged in. This method will ping each UserLoginSession before counting it as a valid login session. The ping protects the system from counting login sessions from GUI's which have been turned off or disconnected without performing a proper logout.

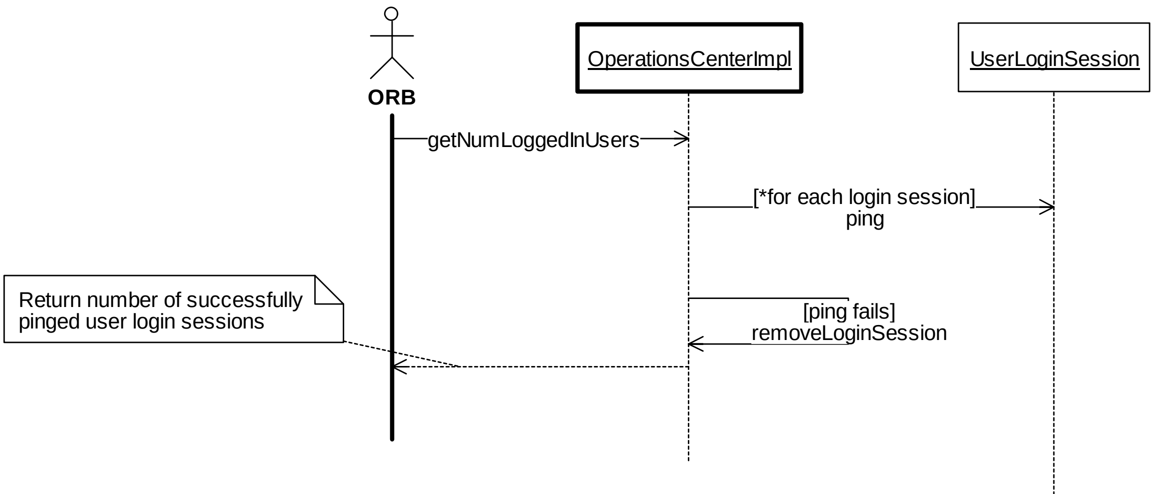

Figure 131. ResourcesModule:GetNumLoggedInUsers (Sequence Diagram)

3.13.2.6 ResourcesModule:Initialize (Sequence Diagram)

When the service is started, the service application will call initialize on this module. The module will create the operations center and organization imlementation objects which are found in the database, connect them to the ORB and export them in the trading service so that other applications may locate them.

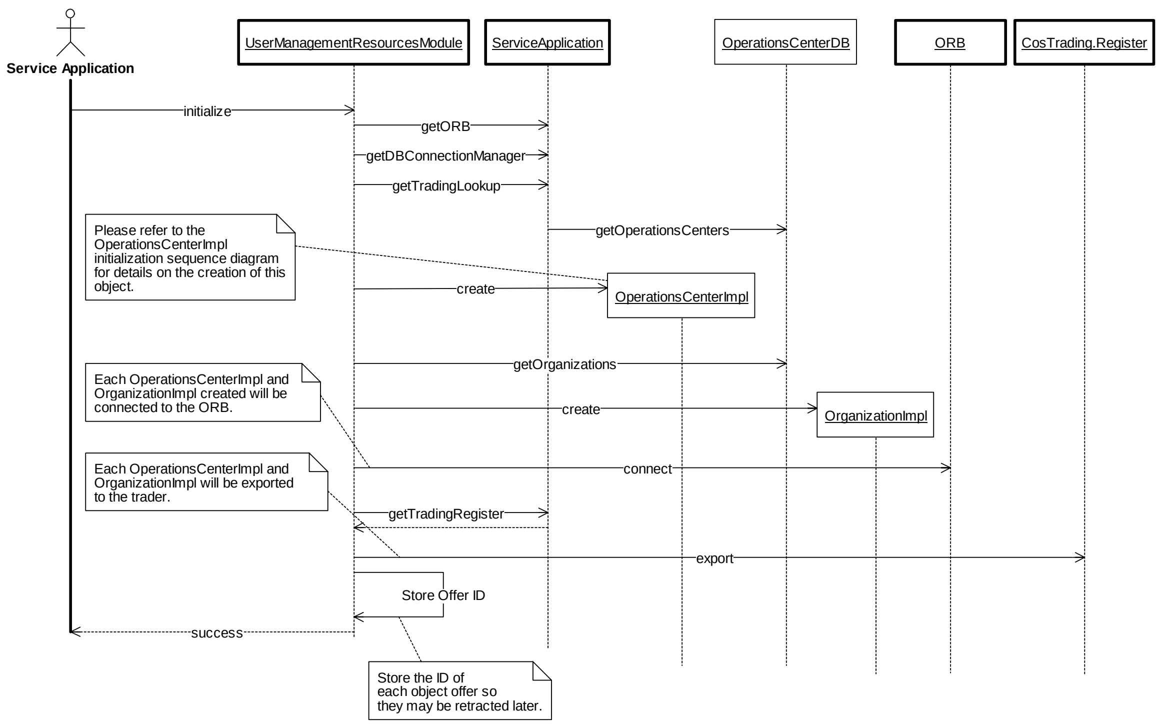

Figure 132. ResourcesModule:Initialize (Sequence Diagram)

3.13.2.7 ResourcesModule:IsUserLoggedIn (Sequence Diagram)

This sequence diagram shows the steps taken to determine if a user is currently logged in to the system.

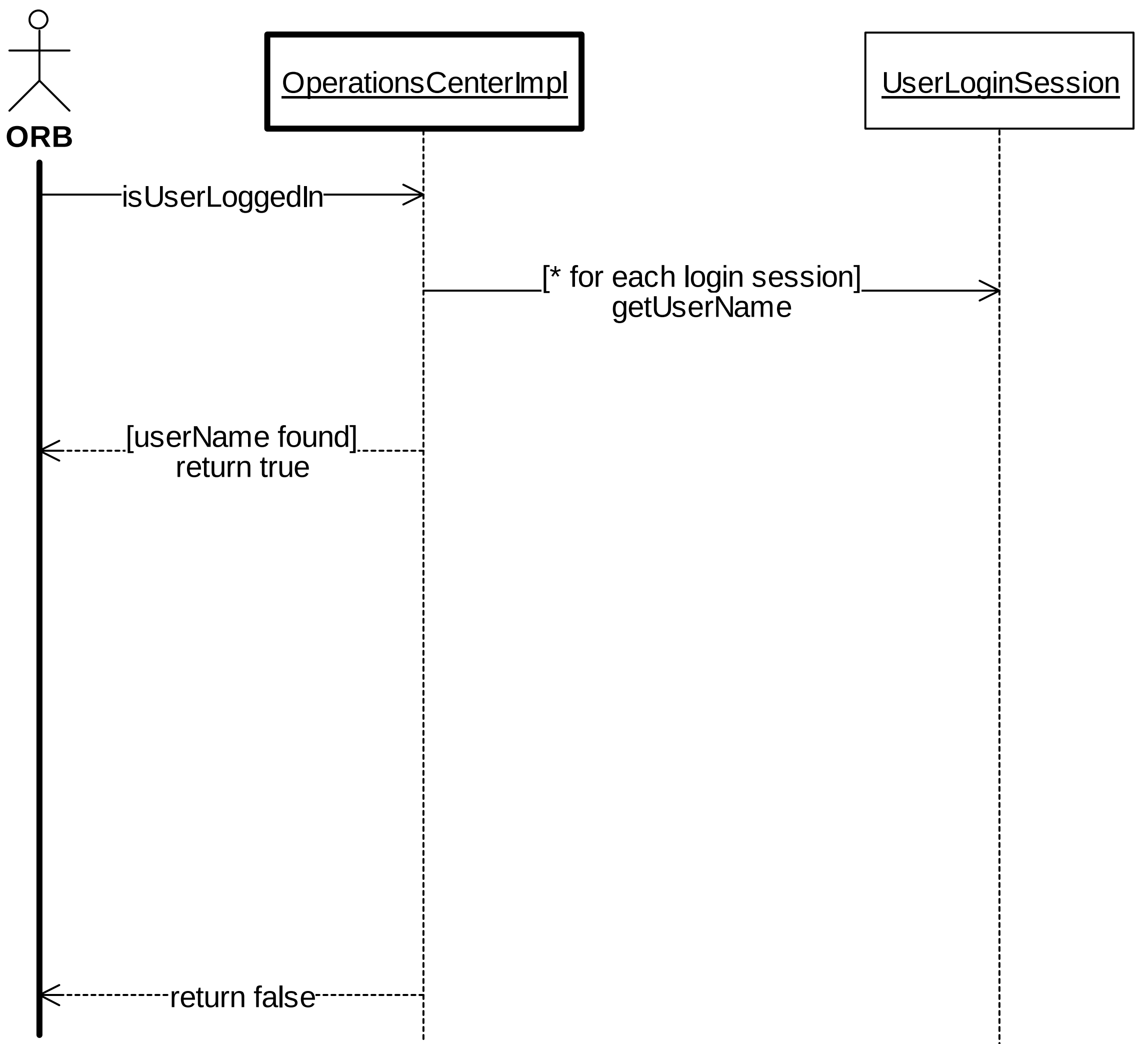

Figure 133. ResourcesModule:IsUserLoggedIn (Sequence Diagram)

3.13.2.8 ResourcesModule:LoginUser (Sequence Diagram)

An client may login to the system. The system will verify that the user has specified the correct password by looking in the user database. If the user has specified the correct password, the system will create a token that contains the user's functional rights and will return it to the invoking client. The login session will be stored internally in the operations center in order to allow the center to respond to calls regarding shared resource control.

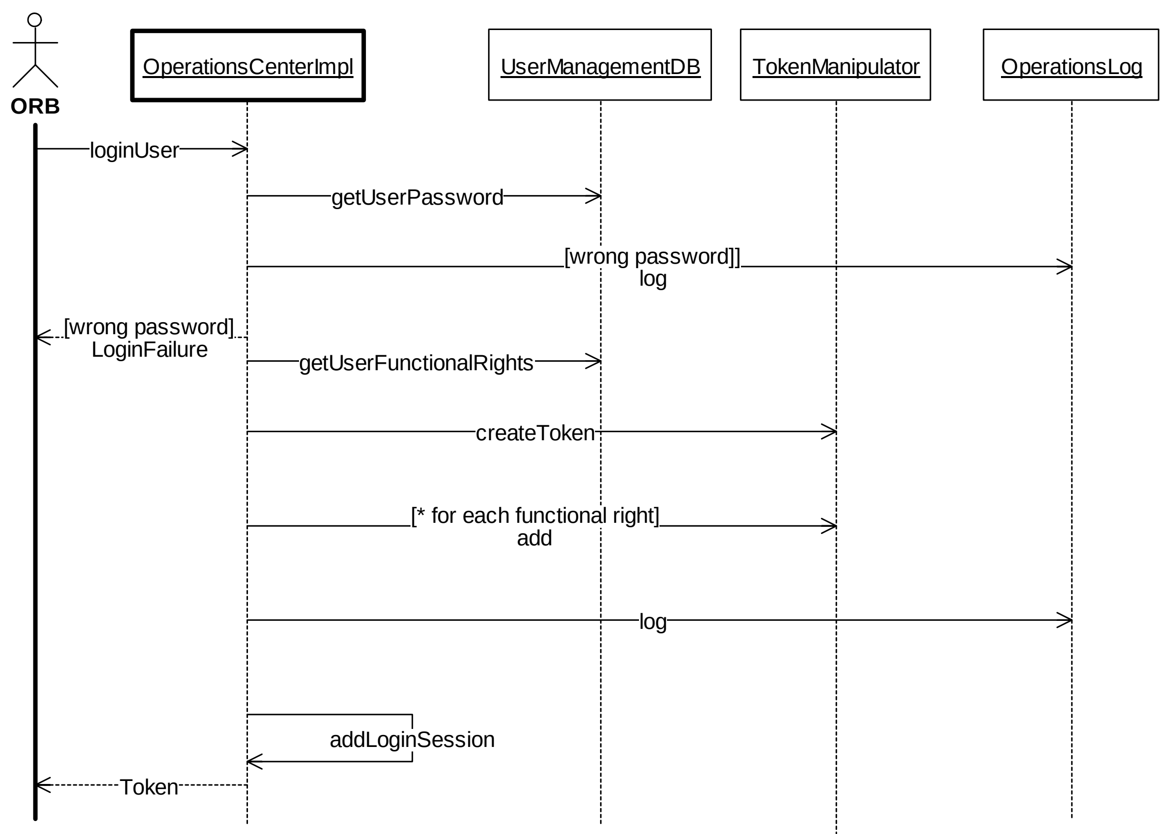

Figure 134. ResourcesModule:LoginUser (Sequence Diagram)

3.13.2.9 ResourcesModule:LogoutUser (Sequence Diagram)

A client may log out of the system. When an operator does this, the system will ping each user login session it is tracking to verify the actual number of users who are currently logged in. If the current number of valid login sessions for this operations center is one, then this user cannot be allowed to logout if this operations center is currently controlling shared resources. In order to determine if the operations center has controlled resources, the system will contact all of the shared resource managers. If the operations center has controlled resources an exception will be thrown, otherwise the user will be logged out.

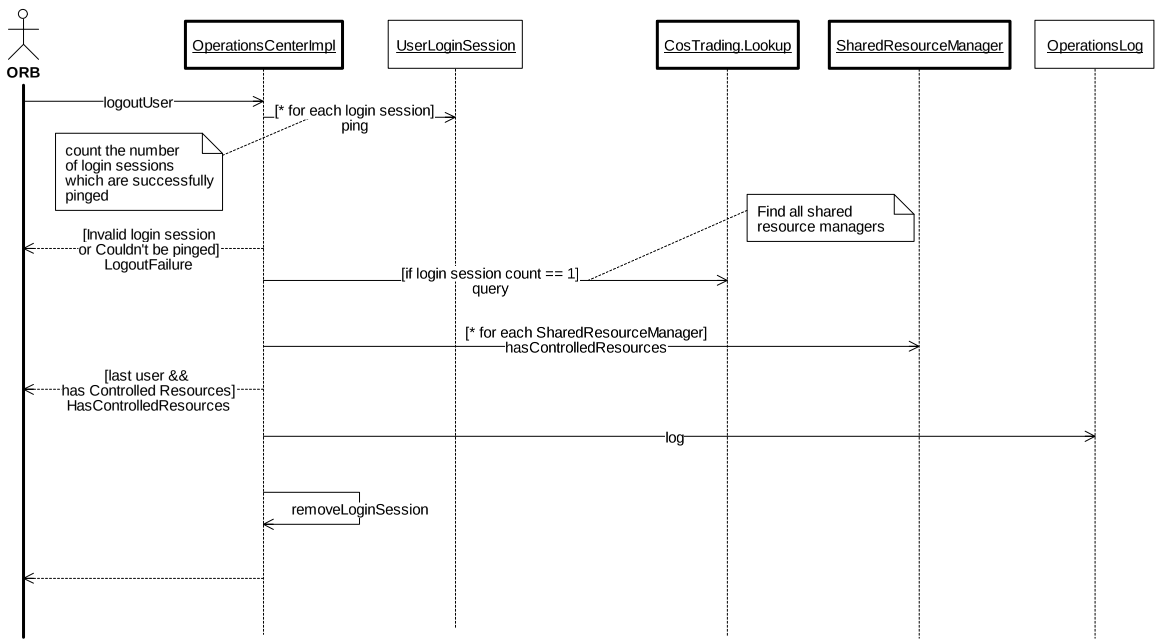

Figure 135. ResourcesModule:LogoutUser (Sequence Diagram)

3.13.2.10 ResourcesModule:OperationsCenterImplInitialization (Sequence Diagram)

This sequence shows the details of constructing an operations center implementation object. An operations center is responsible for tracking the list of currently logged in users. When the service is shutdown it will store the list in the database. When the service is restarted it will get this list of login sessions from the database. Because the service may have been down for an extended period, the login sessions may no longer be valid due to users logging out or shutting down their client machines. Thus, each login session object will be pinged to see if it is still active. If it is, the operations center will add it to the list of current sessions otherwise it will not.

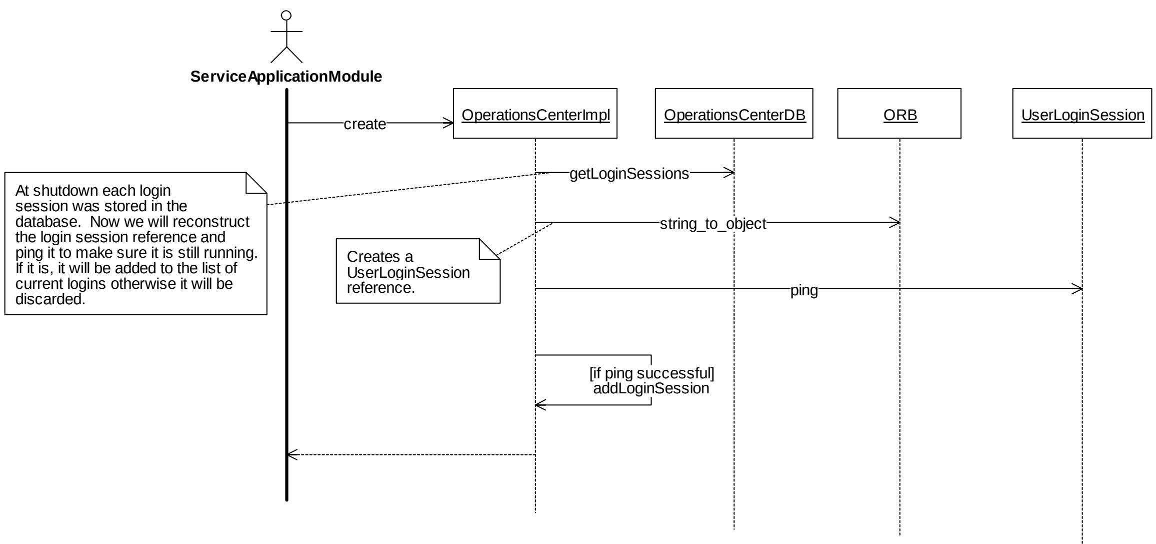

Figure 136. ResourcesModule:OperationsCenterImplInitialization (Sequence Diagram)

3.13.2.11 ResourcesModule:Shutdown (Sequence Diagram)

When the service application calls the shutdown method on this module, the module will withdraw all exported offers from the trader, disconnect any objects that it is currently serving from the ORB and destroy them. The operations center will also store the current list of UserLoginSession references in the database. This will allow the login sessions to be reconstructed at startup.

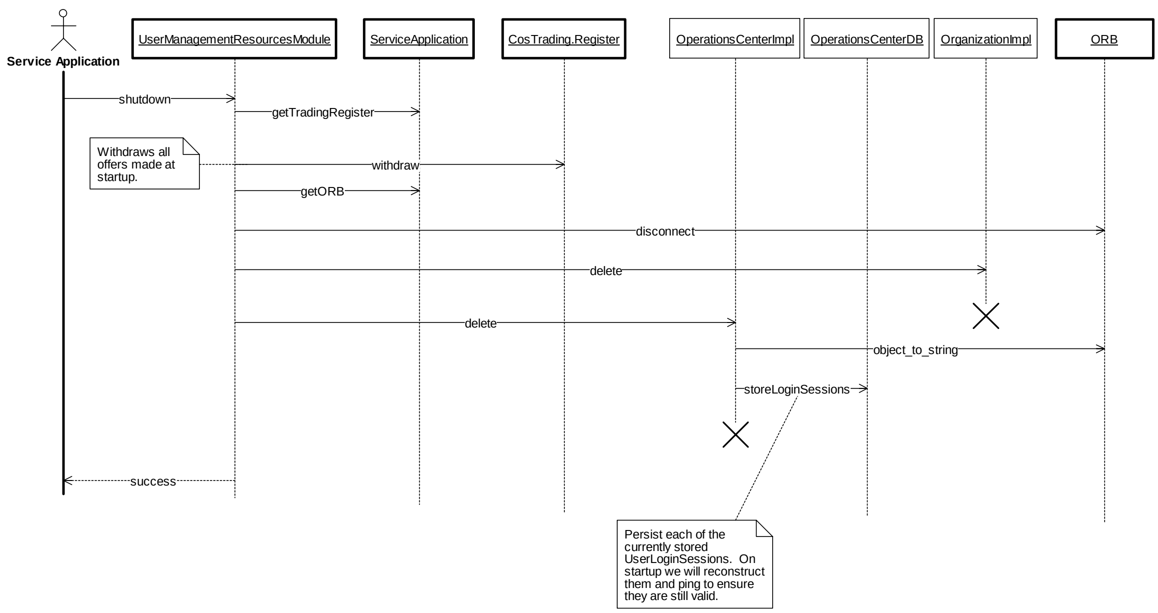

Figure 137. ResourcesModule:Shutdown (Sequence Diagram)

3.13.2.12 ResourcesModule:TransferSharedResources (Sequence Diagram)

A client with the correct functional rights may transfer the control of shared resources from this operations center to another. The system will verify that there are users logged in at the target operations center and will then transfer control of the shared resources if there are.

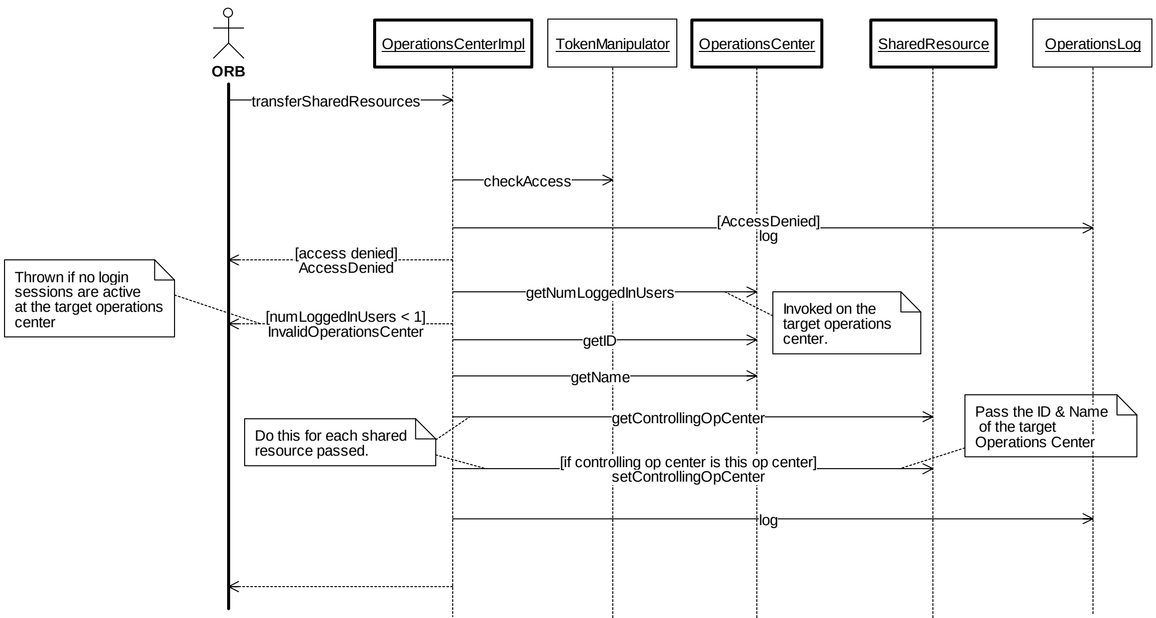

Figure 138. ResourcesModule:TransferSharedResources (Sequence Diagram)

3.14 SHAZAMControl

3.14.1 Classes

3.14.1.1 SHAZAMControl (Class Diagram)

The SHAZAMControlModule serves a SHAZAMFactory object and SHAZAM objects. The class diagram below shows the classes used to implement these system interfaces. Details are only shown for classes in the SHAZAMControlModule package.

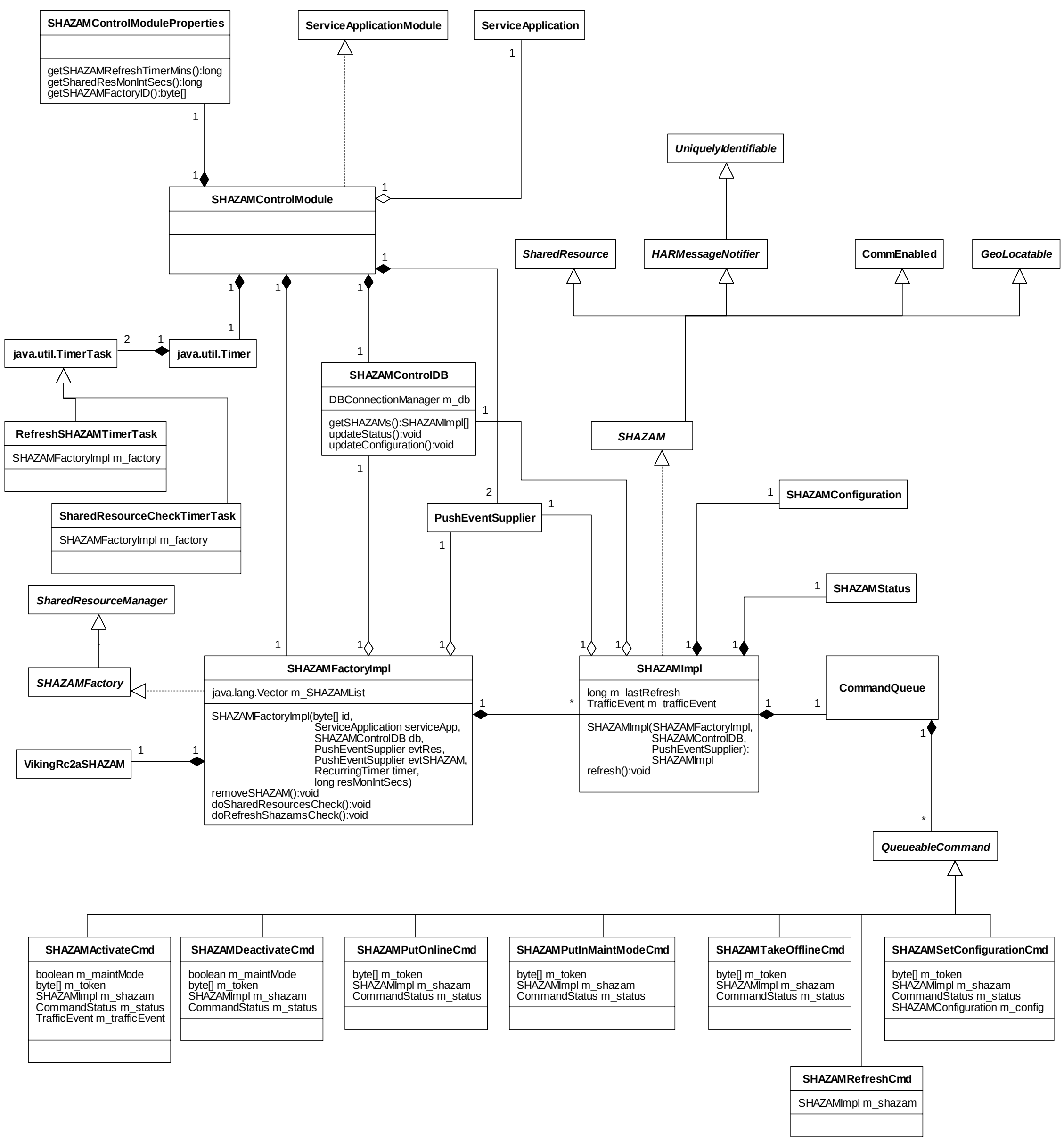

Figure 139. SHAZAMControl (Class Diagram)

3.14.1.1.1 CommandQueue (Class)

The CommandQueue class provides a queue for QueuableCommand objects. The CommandQueue has a thread that it uses to process each QueuableCommand in a first in first out order. As each command object is pulled off the queue by the CommandQueue's thread, the command object's execute method is called, at which time the command performs its intended task.

3.14.1.1.2 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.14.1.1.3 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.14.1.1.4 HARMessageNotifier (Class)

The HARMessageNotifier class specifies an interface to be implemented by devices that can be used to notify the traveler to tune in to a radio station to hear a traffic message being broadcast by a HAR. A HARMessageNotifier is directional and allows users of the device to better determine if activation of the device is warranted for the message being broadcast by the HAR. This interface can be implemented by SHAZAMs and by DMS devices that are allowed to provide a SHAZAM-like message in the absence of any more useful messages to display.

3.14.1.1.5 java.util.Timer (Class)

This class provides asynchronous execution of tasks that are scheduled for one-time or recurring execution.

3.14.1.1.6 java.util.TimerTask (Class)

This class is an abstract base class which can be scheduled with a timer to be executed one or more times.

3.14.1.1.7 QueueableCommand (Class)

A QueueableCommand is an interface used to represent a command that can be placed on a CommandQueue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed. This interface must be implemented by any device command in order that it may be queued on a CommandQueue. The CommandQueue driver calls the execute method to execute a command in the queue and a call to the interuppted method is made when a CommandQueue is shut down.

3.14.1.1.8 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier’s push rate.

3.14.1.1.9 RefreshSHAZAMTimerTask (Class)

This class is a task to be invoked periodically by a timer. When invoked, this class will call a method in the SHAZAMFactoryImpl to have it tell each SHAZAM to refresh if necessary.

3.14.1.1.10 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.14.1.1.11 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.14.1.1.12 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.14.1.1.13 SharedResourceCheckTimerTask (Class)

This class is invoked periodically by a timer. When executed this class calls a method in the SHAZAMFactoryImpl to have it check each shared resource and make sure if it has a controlling op center that the controlling op center has at least one user logged in.

3.14.1.1.14 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn't have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.14.1.1.15 SHAZAM (Class)

This class is used to represent a SHAZAM field device. This class uses a helper class to perform the model specific protocol for device command and control.

3.14.1.1.16 SHAZAMActivateCmd (Class)

This class contains data needed to activate a SHAZAM asynchronously via the CommandQueue. A flag is used to determine if the activation is being performed directly on the device while it is in maintenance mode or if the activation is being processed as an extension of setting a HAR message in response to a traffic event.

3.14.1.1.17 SHAZAMConfiguration (Class)

This class contains data that specifies the configuration of a SHAZAM device.

3.14.1.1.18 SHAZAMControlDB (Class)

This class provides access to database functionality needed to support the SHAZAM and SHAZAMFactory classes. This class provides a high level interface to allow for persistence and depersistance of SHAZAM and SHAZAMFactory objects.

3.14.1.1.19 SHAZAMControlModule (Class)

This class is a service module that provides control of SHAZAM devices. Upon initialization the module initializes a SHAZAMFactory which contains SHAZAM objects that have been previously added to the system. These objects are accessed via the CORBA ORB and manipulated directly from client applications. The module also creates support objects that are used by the SHAZAM (and SHAZAMFactory) objects to perform their processing, such as a database connection, event channels, and a periodic timer used to allow the objects to perform timer based processing.

3.14.1.1.20 SHAZAMControlModuleProperties (Class)

This class is used to access SHAZAMControlModule specific settings in the application service's properties file.

3.14.1.1.21 SHAZAMDeactivateCmd (Class)

This class contains data needed to deactivate a SHAZAM asynchronously via the CommandQueue. A flag is used to determine if the deactivation is being performed directly on the device while it is in maintenance mode or if the deactivation is being processed as an extension of setting a HAR message in response to a traffic event.

3.14.1.1.22 SHAZAMFactory (Class)

This CORBA interface allows new SHAZAM objects to be added to the system.

3.14.1.1.23 SHAZAMFactoryImpl (Class)

This class provides the ability to add new SHAZAM objects to the system. When SHAZAMs are added, they are persisted to the database so this object can depersist them upon startup. This class also provides a removeSHAZAM method that allows a SHAZAM to remove itself from the system when directed.

3.14.1.1.24 SHAZAMImpl (Class)

This class implements the SHAZAM interface and allows for control of a SHAZAM field device. The SHAZAMImpl makes use of the VikingRc2aSHAZAM object to perform field communications to the device. All field communications are done asynchronously via the command queue thread. The progress of an asynchronous command is provided to the caller via a CommandStatus object.

3.14.1.1.25 SHAZAMPutInMaintModeCmd (Class)

This command contains data needed to put a SHAZAM device in maintenance mode asynchronously via the CommandQueue. When executed this class calls back into the SHAZAMImpl object to perform the "put in maintenance mode"

3.14.1.1.26 SHAZAMPutOnlineCmd (Class)

This command contains data needed to put a SHAZAM device online asynchronously via the CommandQueue. When executed this class calls back into the SHAZAMImpl object to perform the "put online" processing.

3.14.1.1.27 SHAZAMRefreshCmd (Class)

This class is a command object used to invoke the SHAZAM refresh processing asynchronously from the command queue.

3.14.1.1.28 SHAZAMSetConfigurationCmd (Class)

This command contains data needed to put set the SHAZAM configuration asynchronously via the CommandQueue. When executed this class calls back into the SHAZAMImpl object to perform the “set configuration” processing. The SHAZAM device model currently in use does not contain any configuration settings, however this command is still processed asynchronously for consistency.

3.14.1.1.29 SHAZAMStatus (Class)

This class contains the current status of a SHAZAM device.

3.14.1.1.30 SHAZAMTakeOfflineCmd (Class)

This command contains data needed to take a SHAZAM device offline asynchronously via the CommandQueue. When executed this class calls back into the SHAZAMImpl object to perform the “take offline” processing.

3.14.1.1.31 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.14.1.1.32 VikingRc2aSHAZAM (Class)

This class provides the device specific prototcol for controlling a SHAZAM device. This class uses a TelephonyManager to acquire a telephony port for processing. It then uses the telephony port to connect to the SHAZAM and send DTMF to activate or deactivate the beacons via the Viking controller.

This class is responsible for intelligence in acquiring a port, such as seeking out an alternate TelephonyManager when necessary.

3.14.2 Sequence Diagrams

3.14.2.1 SHAZAMControlModule:activateSHAZAM (Sequence Diagram)

A SHAZAM can be activated by a HAR when its message is set, or it can be activated directly when in maintenance mode. In either case, the processing done is nearly identical. When being activated by a HAR as part of the HAR message activation, the activateHARNotice method from the HARMessageNotifier interface is called. When being activated directly, the SHAZAM's setBeaconsOn method is called.

Regardless of the API called, the SHAZAM creates a SHAZAMActivateCmd object and places it on its command queue for asynchronous processing. A flag in the SHAZAMActivateCmd object specifies the activation was requested from maintenance mode or online mode. When the queue executes the command, the activateImpl method checks the flags in the command object to determine any processing that is specific to the mode in which the activation request occurred. Common processing includes calling the VikingRc2aSHAZAM object to perform communications and command the SHAZAM and utilizing the caller's command status object to inform the caller of the command's progress. Specific processing that requires checking the mode of the request includes checking that the SHAZAM is in the same mode as when the command was queued, and updating the TrafficEvent's history if the activation occurred in online mode.

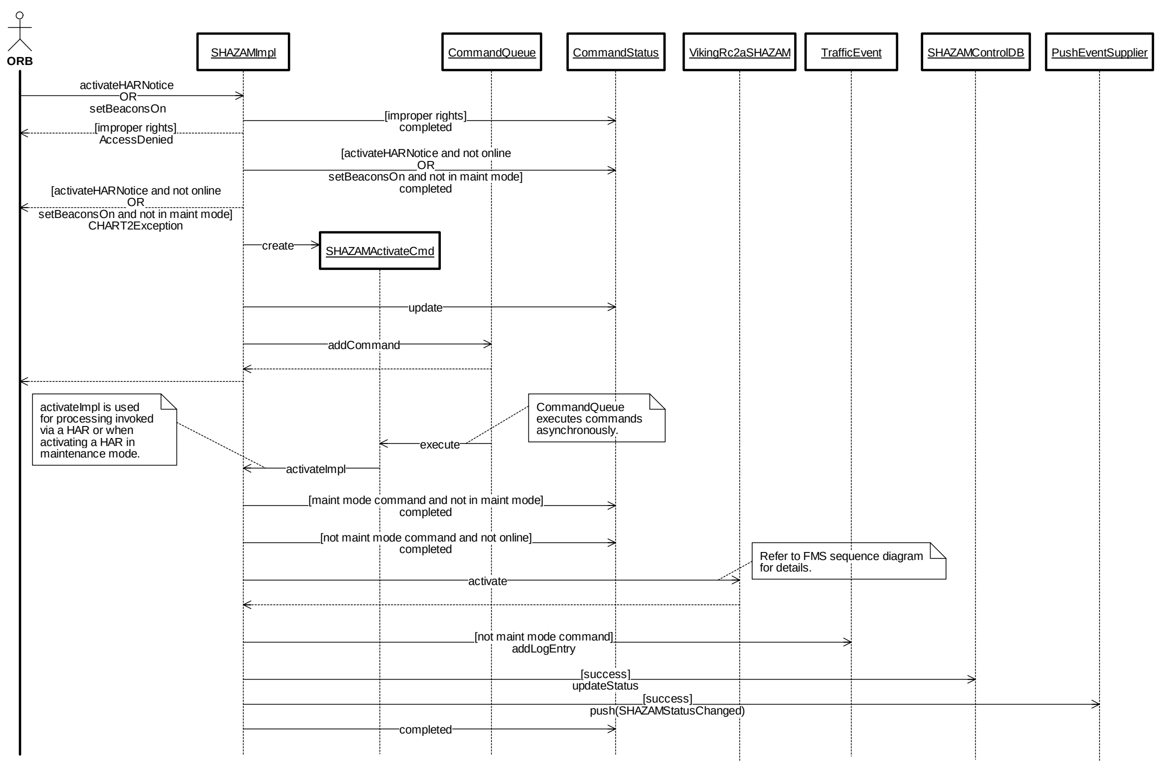

Figure 140. SHAZAMControlModule:activateSHAZAM (Sequence Diagram)

3.14.2.2 SHAZAMControlModule:createSHAZAM (Sequence Diagram)

A user with the proper functional rights can add a SHAZAM to the system. The SHAZAM configuration data is added to the database, a SHAZAMImpl object is created, and the object is connected to the POA, making it ready for calls from clients. The ServiceApplication is called to register the object with the trader and an event is pushed to allow GUIs to show this SHAZAM as an available object in the system. The SHAZAM is added in the offline state and no field communications are necessary.

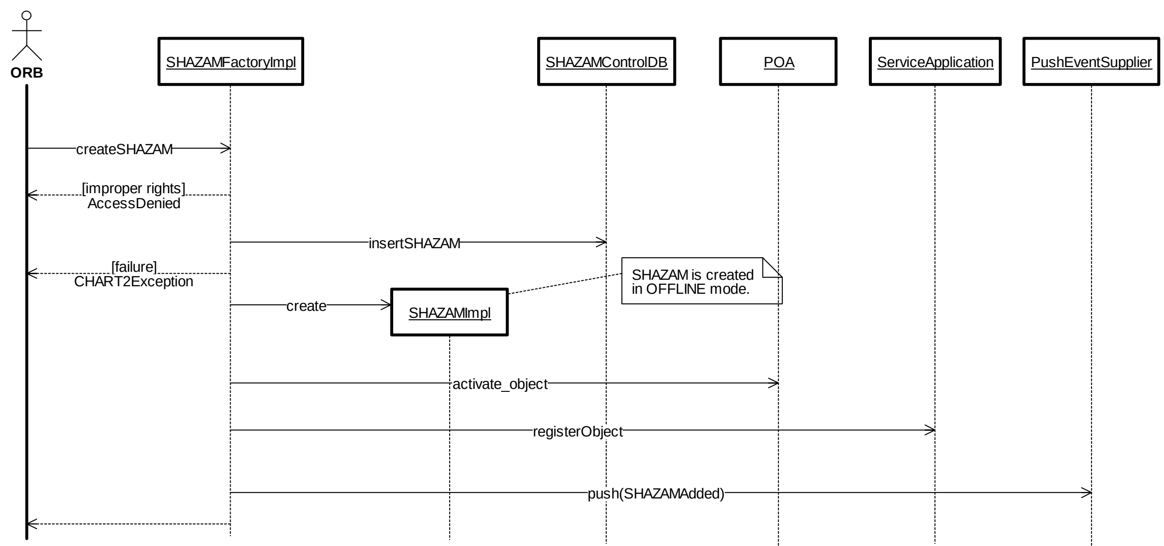

Figure 141. SHAZAMControlModule:createSHAZAM (Sequence Diagram)

3.14.2.3 SHAZAMControlModule:deactivateSHAZAM (Sequence Diagram)

A SHAZAM can be deactivated by a HAR when its message is set, or it can be deactivated directly when in maintenance mode. In either case, the processing done is nearly identical. When being deactivated by a HAR as part of the HAR message activation/blank processing, the deactivateHARNotice method from the HARMessageNotifier interface is called. When being deactivated directly, the SHAZAM's setBeaconsOff method is called.

Regardless of the API called, the SHAZAM creates a SHAZAMDeactivateCmd object and places it on its command queue for asynchronous processing. A flag in the SHAZAMDeactivateCmd object specifies the deactivation was requested from maintenance mode or online mode. When the queue executes the command, the deactivateImpl method checks the flags in the command object to determine any processing that is specific to the mode in which the deactivation request occurred. Common processing includes calling the VikingRc2aSHAZAM object to perform communications and command the SHAZAM and utilizing the caller's command status object to inform the caller of the command's progress. Specific processing that requires checking the mode of the request includes checking that the SHAZAM is in the same mode as when the command was queued, and updating the TrafficEvent's history if the deactivation occurred in online mode.

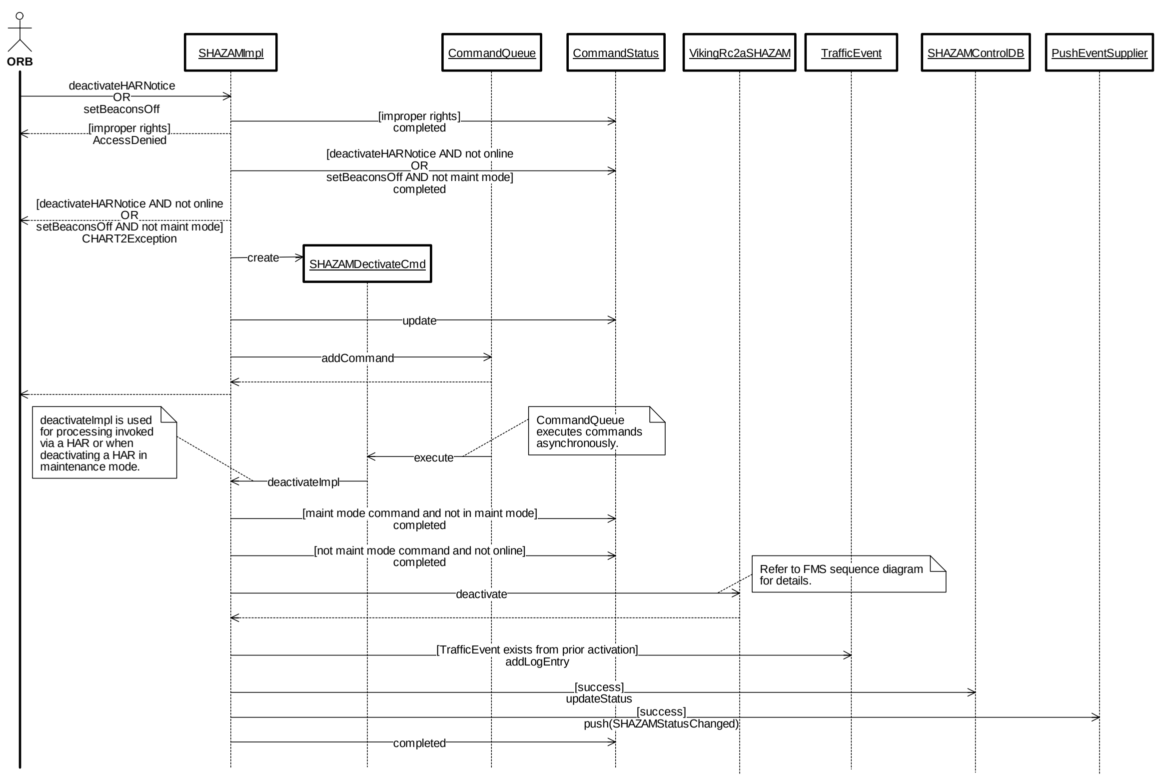

Figure 142. SHAZAMControlModule:deactivateSHAZAM (Sequence Diagram)

3.14.2.4 SHAZAMControlModule:initialize (Sequence Diagram)

When the SHAZAMControlModule is included in a ServiceApplication, the service application calls the SHAZAMControlModule's initialize method when the service is started. The SHAZAMControlModule creates supporting objects such as the SHAZAMControlModuleDB for database access and PushEventSupplier objects for resource management events and SHAZAM control events. A SHAZAMFactoryImpl object is created which depersists all SHAZAMs that have been previously added to the system. Each SHAZAM is connected to the ORB and registered with the service application to have the object published in the trader. A Timer is used to call the SHAZAMFactory to perform timer based processing.

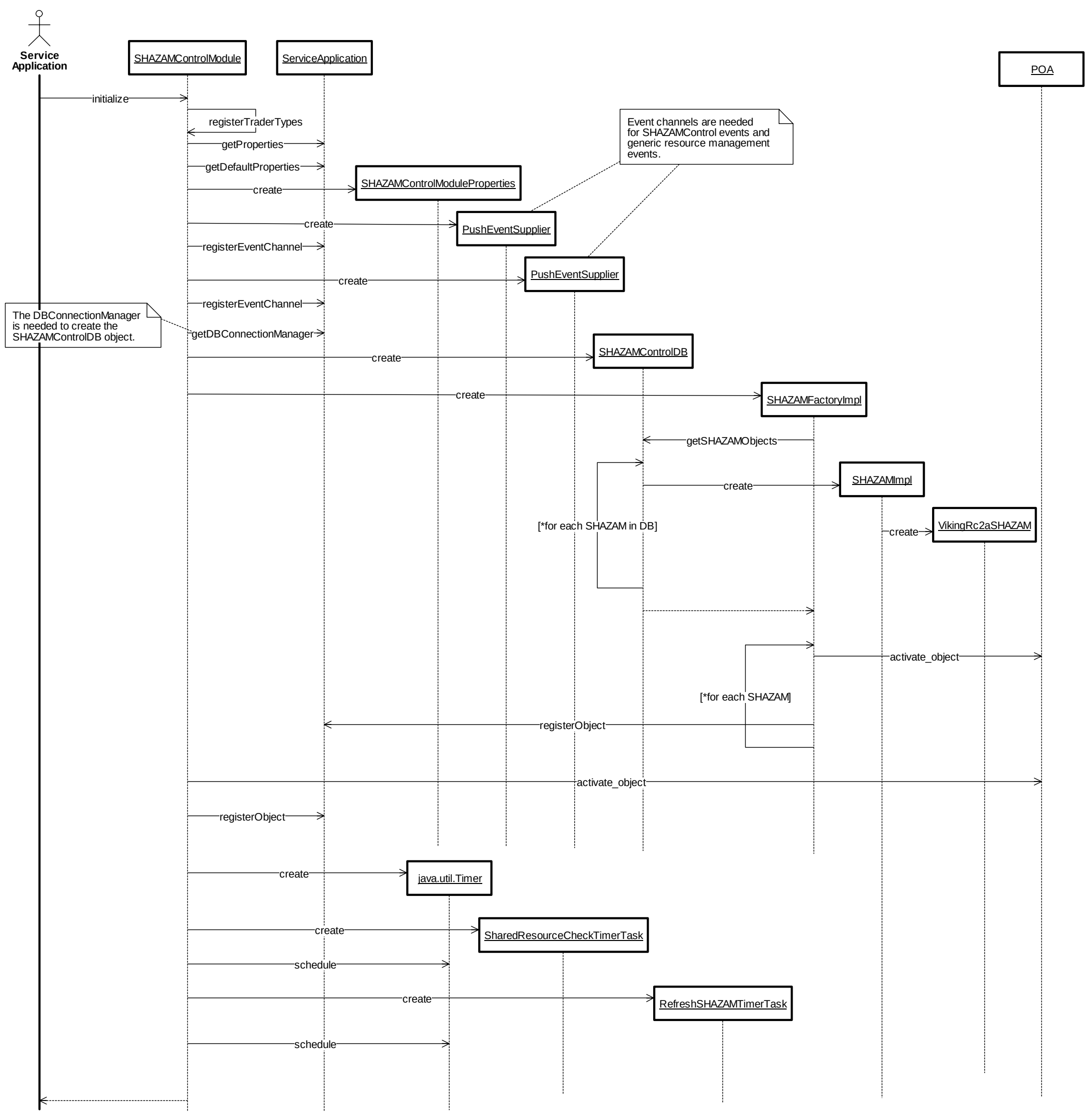

Figure 143. SHAZAMControlModule:initialize (Sequence Diagram)

3.14.2.5 SHAZAMControlModule:putInMaintenanceMode (Sequence Diagram)

A user with proper functional rights can put a SHAZAM in maintenance mode if it is not already in maintenance mode. A command object is created and placed on the command queue to execute the command asynchronously. When executed, the command calls back into the SHAZAMImpl object that calls the VikingRc2aSHAZAM object to command the device to its inactive state. Regardless of the ability to command the device, the SHAZAMImpl changes to the maintenance mode state, pesists its state in the database, and pushes an event to allow the GUI to update its display for the SHAZAM.

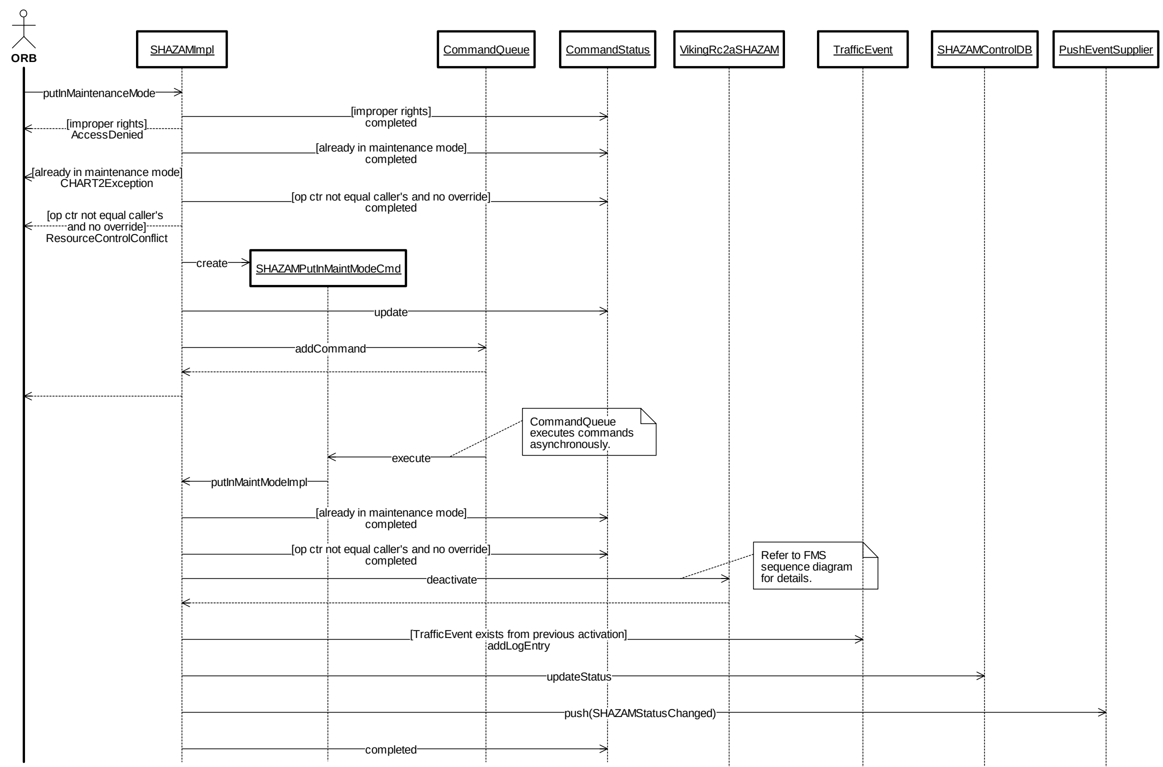

Figure 144. SHAZAMControlModule:putInMaintenanceMode (Sequence Diagram)

3.14.2.6 SHAZAMControlModule:putOnline (Sequence Diagram)

A user with proper functional rights can put a SHAZAM online if it is not already online. A command object is created and placed on the command queue to execute the command asynchronously. When executed, the command calls back into the SHAZAMImpl object that calls the VikingRc2aSHAZAM object to command the device to a known state (not active). If able to deactivate the device, the SHAZAMImpl changes to the online state, pesists its state in the database, and pushes an event to allow the GUI to update its display for the SHAZAM.

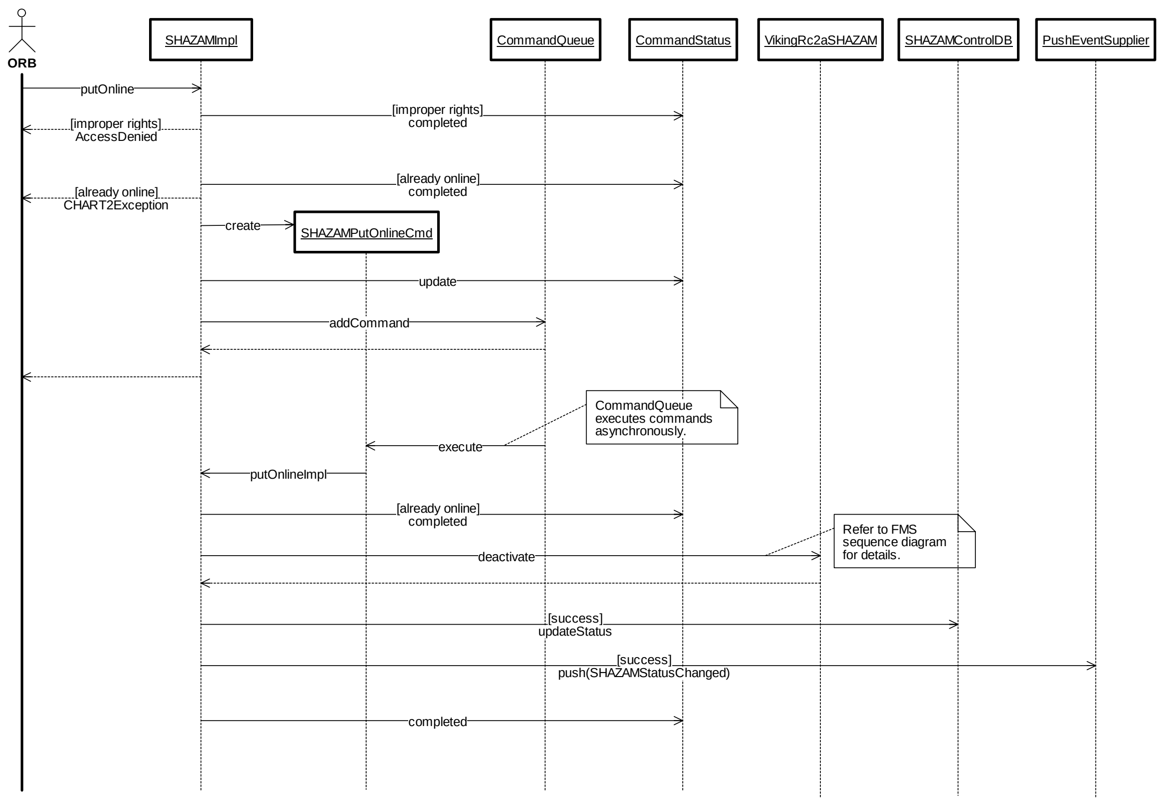

Figure 145. SHAZAMControlModule:putOnline (Sequence Diagram)

3.14.2.7 SHAZAMControlModule:remove (Sequence Diagram)

A user with the proper functional rights can remove an offline SHAZAM from the system. The SHAZAM object is withdrawn from the trader and disconnected from the ORB. The data for the SHAZAM is deleted from the database and a message is pushed to allow the GUIs to remove the SHAZAM.

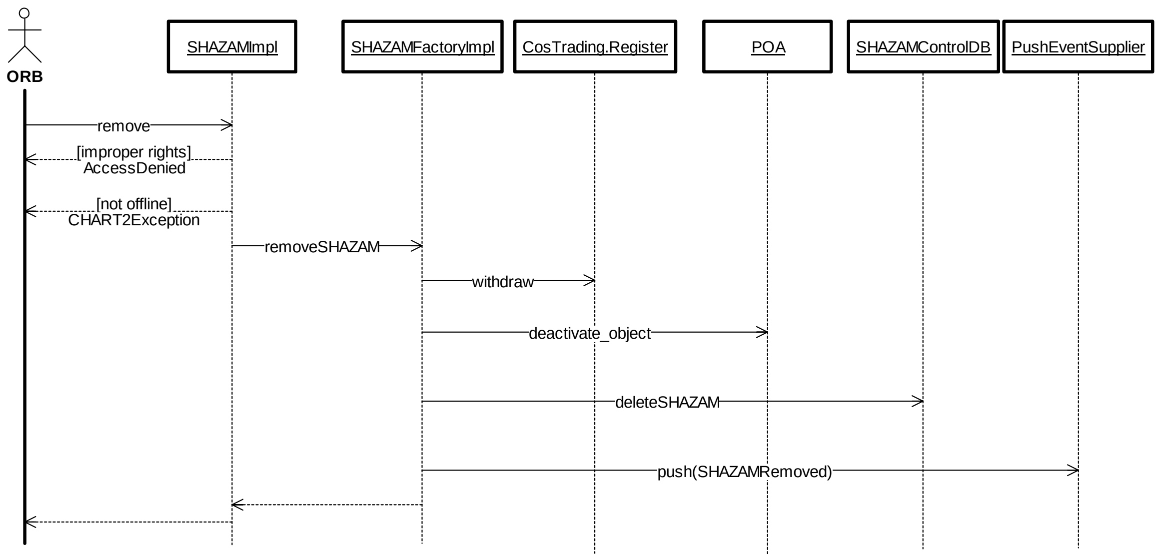

Figure 146. SHAZAMControlModule:remove (Sequence Diagram)

3.14.2.8 SHAZAMControlModule:ResetSHAZAMtoLastKnownState (Sequence Diagram)

Because SHAZAMs do not issue any response to commands and these devices have been found to be less than reliable in the past, a process is in place to periodically command the device to its last known status. A Timer notifies the SHAZAMRefreshTimerTask when the task's scheduled interval expires. The task calls the SHAZAMFactoryImpl which calls each SHAZAM to have them do a refresh if necessary. Each SHAZAM determines if a refresh is necessary based on its refresh interval. Refreshes are only done when the SHAZAM is in an online state. If the SHAZAM determines a refresh is warranted it adds a refresh commandto its command queue to be executed asynchronously. When the command is executed, it makes sure the refresh is still necessary and the appropriate command (activate or deactivate) is sent to the device via the VikingRc2aSHAZAM class. A low priority is given to the command in terms of communications resource usage. (Refer to the FMS detailed design for more information on communications resources and priorities)

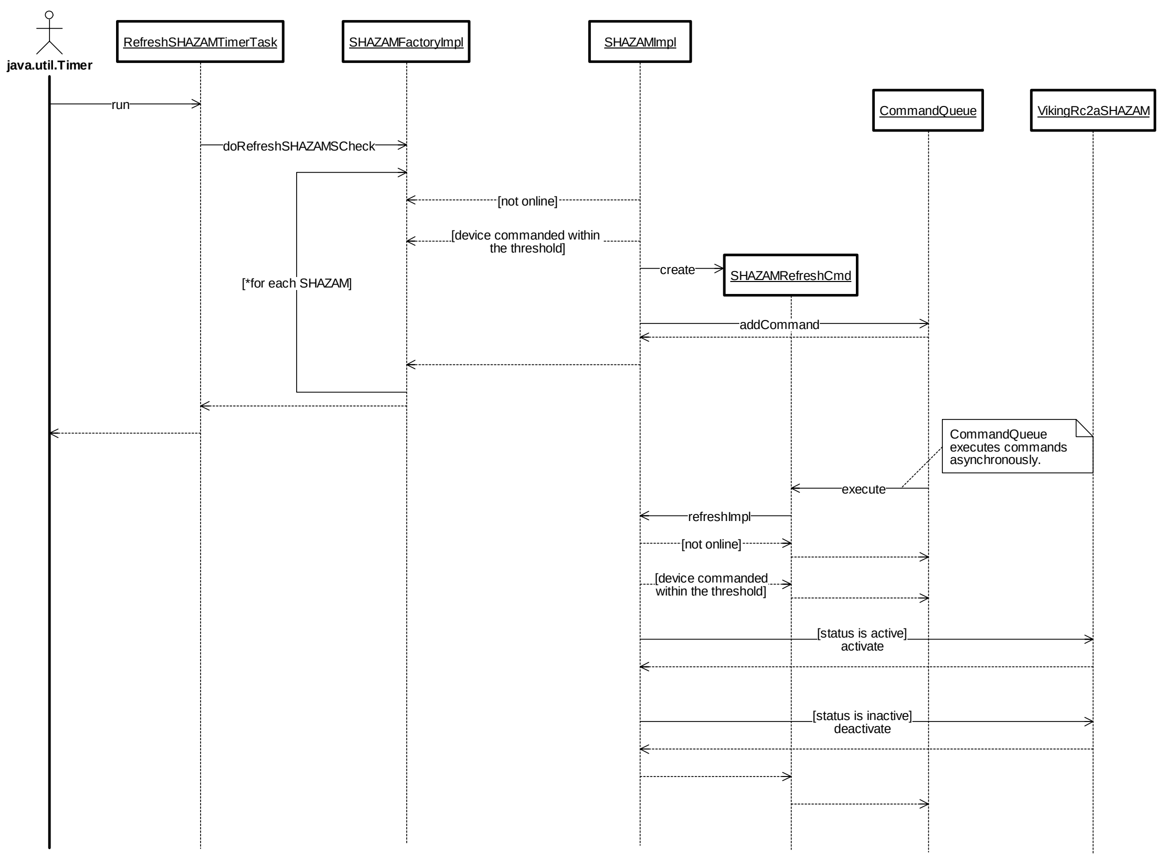

Figure 147. SHAZAMControlModule:ResetSHAZAMtoLastKnownState (Sequence Diagram)

3.14.2.9 SHAZAMControlModule:setConfiguration (Sequence Diagram)

A user with appropriate functional rights can set the configuration of a SHAZAM if it is in maintenance mode. The Rc2aSHAZAM itself does not have any configurable settings, so no field communications are necessary. The configuration is stored in memory and persisted to the database and an event is pushed to notify others of the changes. Note that although this command does not currently require field communications, the asynchronous command pattern is used for consistency with other device commands and also to allow the code to easily adapt to a device type that supports configurable settings.

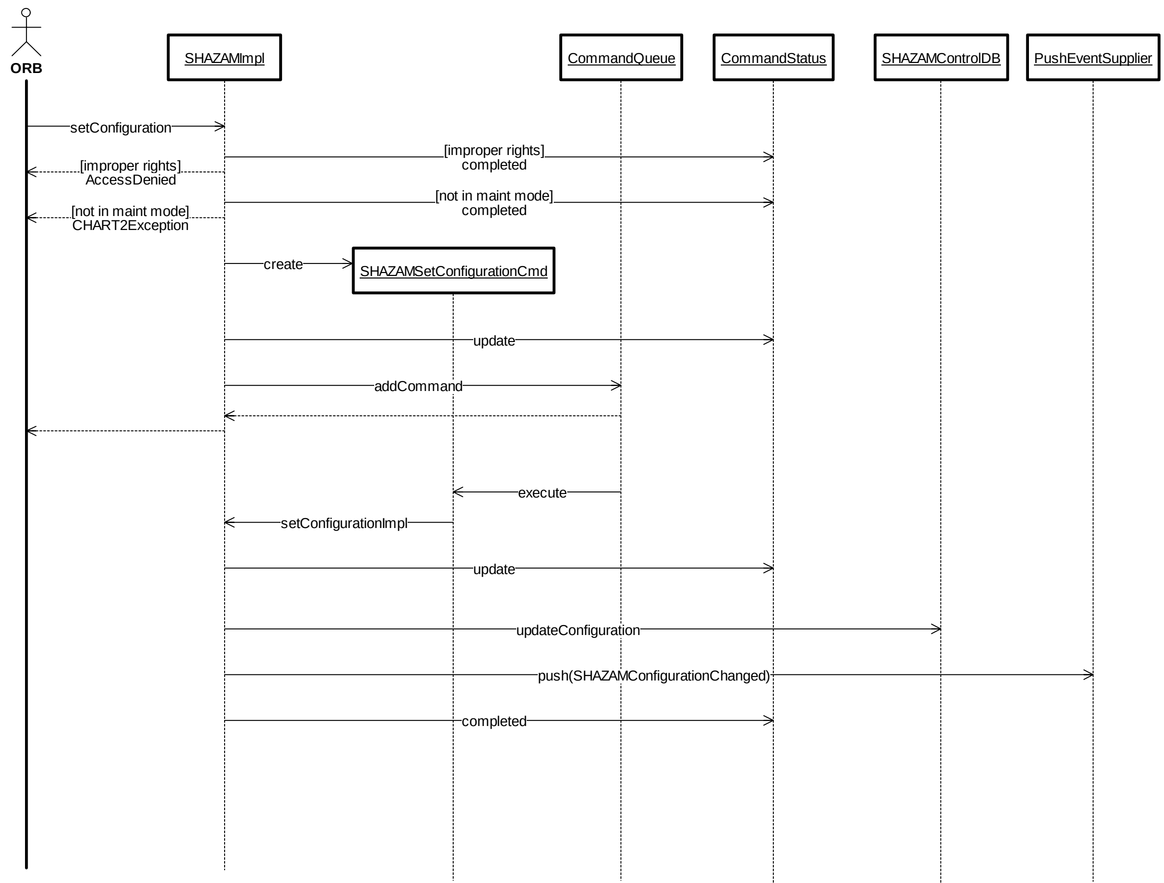

Figure 148. SHAZAMControlModule:setConfiguration (Sequence Diagram)

3.14.2.10 SHAZAMControlModule:shutdown (Sequence Diagram)

When a service application containing the SHAZAMControlModule is shutdown, it calls the shutdown method. The SHAZAMControlModule cleans up its resources, which include its periodic timer and PushEventConsumers.

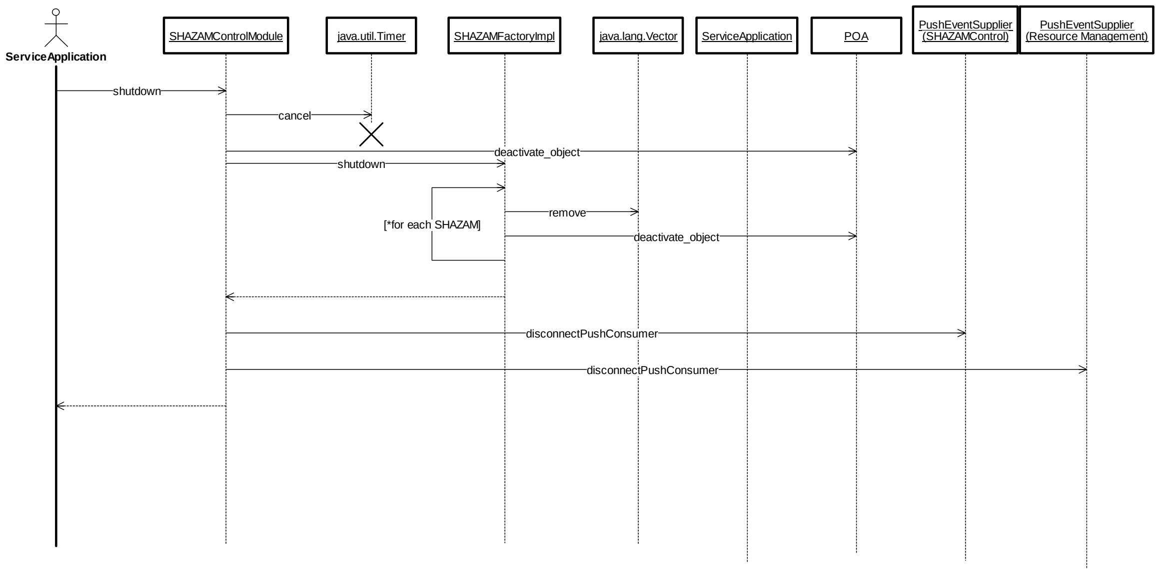

Figure 149. SHAZAMControlModule:shutdown (Sequence Diagram)

3.14.2.11 SHAZAMControlModule:takeOffline (Sequence Diagram)

A user with proper functional rights can take a SHAZAM offline if it is not already offline. A command object is created and placed on the command queue to execute the command asynchronously. When executed, the command calls back into the SHAZAMImpl object that calls the VikingRc2aSHAZAM object to command the device to its inactive state. Regardless of the ability to command the device, the SHAZAMImpl changes to the offline state, pesists its state in the database, and pushes an event to allow the GUI to update its display for the SHAZAM.

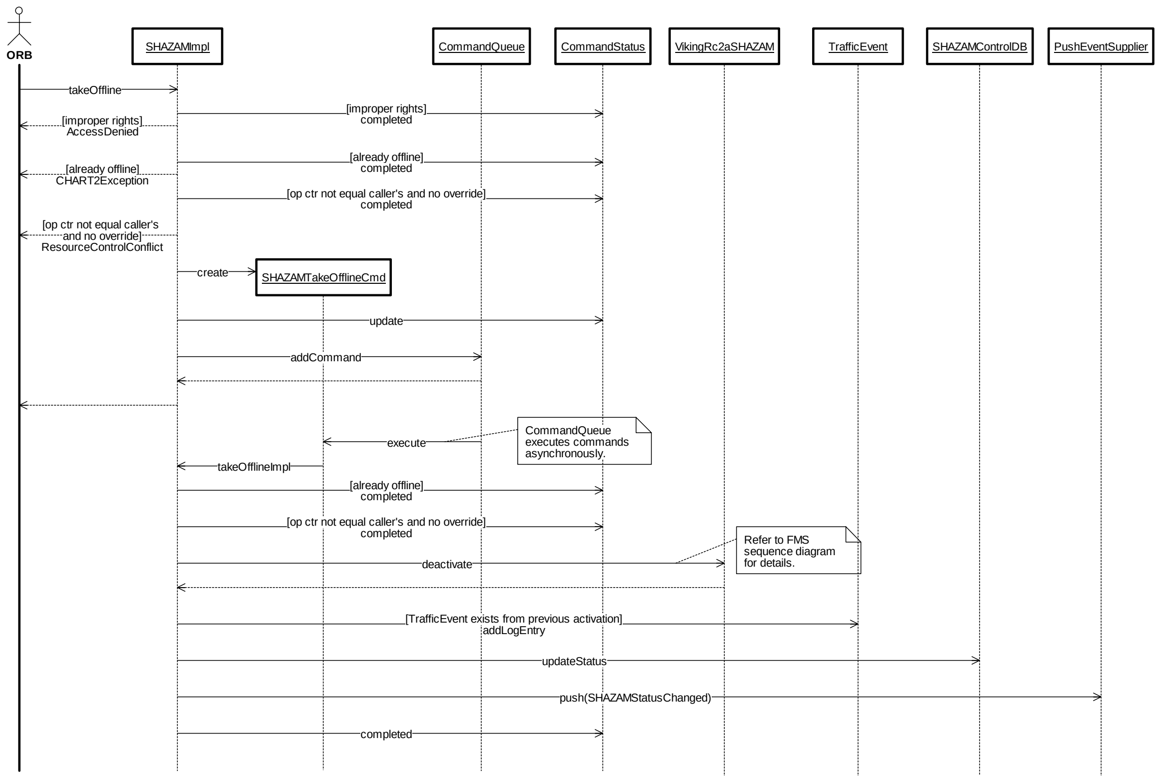

Figure 150. SHAZAMControlModule:takeOffline (Sequence Diagram)

3.15 SHAZAMUtility

3.15.1 Classes

3.15.1.1 SHAZAMUtility (Class Diagram)

This diagram shows SHAZAM related classes that are shared between the server and the GUI.

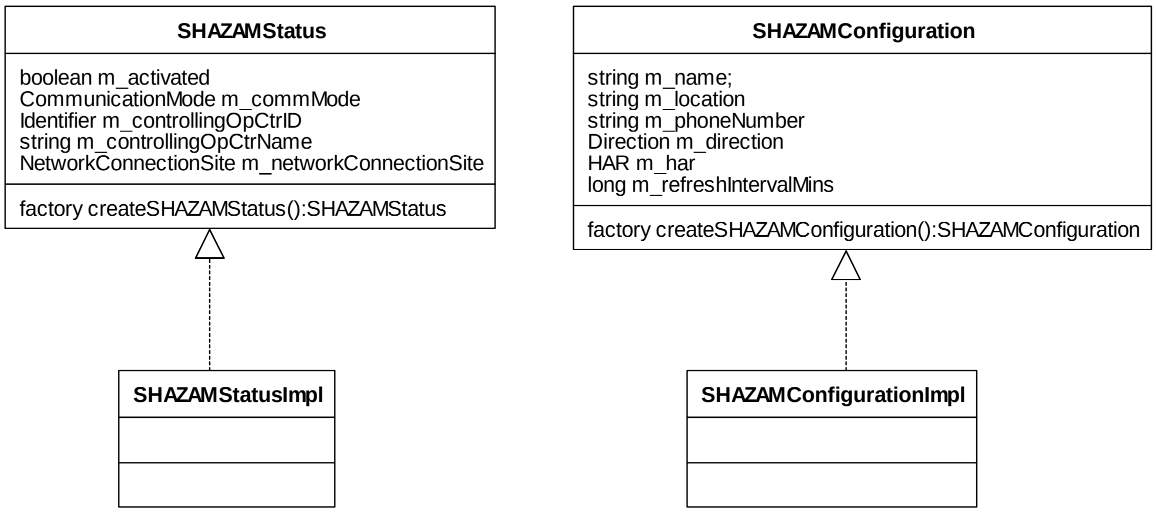

Figure 151. SHAZAMUtility (Class Diagram)

3.15.1.1.1 SHAZAMConfiguration (Class)

This class contains data that specifies the configuration of a SHAZAM device.

3.15.1.1.2 SHAZAMConfigurationImpl (Class)

This class provides an implementation of the SHAZAMConfiguration valuetype as defined in the IDL. This class provides access to values relating to the configuration of a SHAZAM.

3.15.1.1.3 SHAZAMStatus (Class)

This class contains the current status of a SHAZAM device.

3.15.1.1.4 SHAZAMStatusImpl (Class)

This class implements the SHAZAMStatus valuetype as defined in the IDL. It provides access to values relating to the current status of a SHAZAM.

3.16 SystemInterfaces

This section shows interfaces to the system that are defined in IDL.

3.16.1 Classes

3.16.1.1 AudioCommon (Class Diagram)

This class diagram shows the classes relating to Audio.

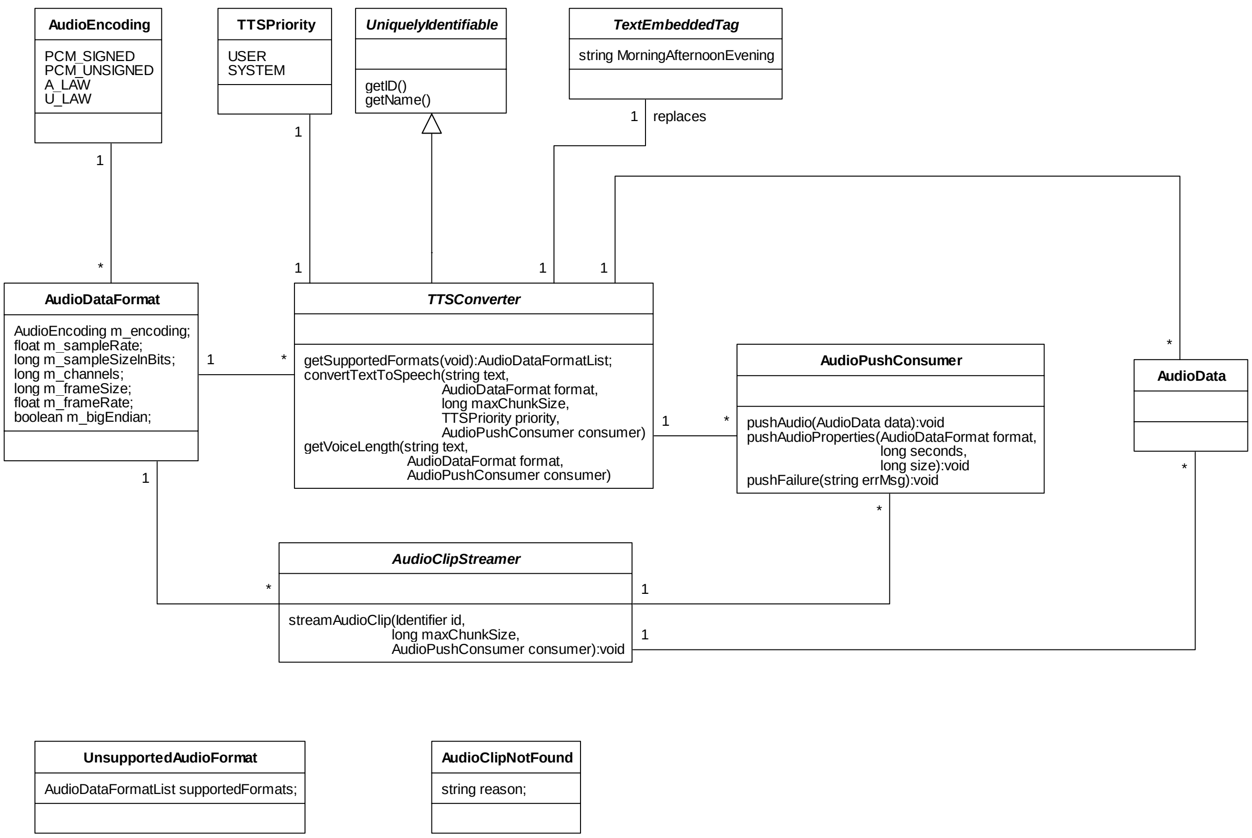

Figure 152. AudioCommon (Class Diagram)

3.16.1.1.1 AudioClipNotFound (Class)

This exception is thrown by an AudioClipStreamer if asked to push an audio clip which it cannot find.

3.16.1.1.2 AudioClipStreamer (Class)

This interface is implemented by objects that can push a previously stored audio clip given its ID. The audio data is pushed via the AudioPushConsumer supplied by the user of this interface.

3.16.1.1.3 AudioData (Class)

This typedef is a sequence of bytes that contain audio data. This data is used in conjunction with AudioDataFormat to decode the data into voice.

3.16.1.1.4 AudioDataFormat (Class)

This struct specifies the format of audio data.

3.16.1.1.5 AudioEncoding (Class)

This enum defines the supported types of encoding for audio data.

3.16.1.1.6 AudioPushConsumer (Class)

This interface is implemented by objects that may need to receive audio data using the push model, where the server pushes the data to the consumer. One call to pushAudioProperties() will always precede any calls to pushAudio().

3.16.1.1.7 TextEmbeddedTag (Class)

This interface defines constants for tags that may be embedded in text that is passed to the TTSConverter. The TTSConverter replaces the tags it finds in text prior to converting the text to speech. The MorningAfternoonEvening tag is replaced with the text ‘morning’ when the conversion takes place between 00:00 and 11:59, ‘afternoon’ from 12:00 through 16:59, and ‘evening’ from 17:00 to 23:59.

3.16.1.1.8 TTSConverter (Class)

This interface represents the Text to Speech converter object that allows text to be passed in and speech to be returned.

3.16.1.1.9 TTSPriority (Class)

This enum defines the types of priorities that can be used when asking the TTSConverter to convert text to speech.

3.16.1.1.10 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.16.1.1.11 UnsupportedAudioFormat (Class)

This exception is thrown when a specific AudioDataFormat is requested from an object that does not support the given format.

3.16.1.2 CommLogManagement (Class Diagram)

This Class Diagram shows the classes used for passing information between processes to enable creating, pushing, viewing, and searching Communications Log entries.

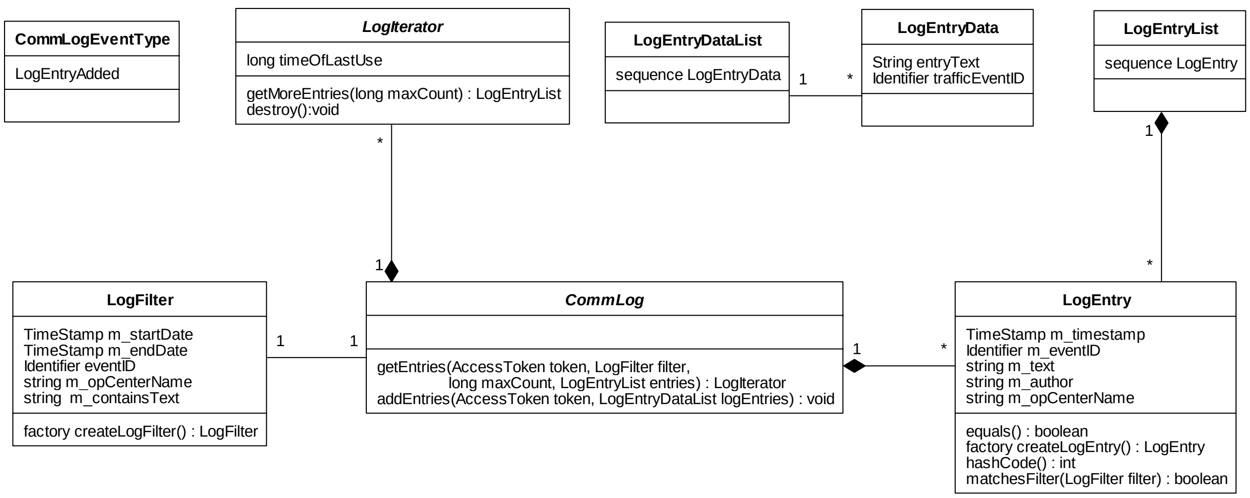

Figure 153. CommLogManagement (Class Diagram)

3.16.1.2.1 CommLog (Class)

This class manages log entries. These can be general Communications Log entries or specific log entries for a specific Traffic Event. This class is the primary interface for the CommLog service. It is used to persist log entries in the CHART II system and retrieve them for review. Log entries can be created directly by users or indirectly as a result of manipulating Traffic Events.

3.16.1.2.2 CommLogEventType (Class)

This enumeration lists the possible events that the CommsLog service may push via the CORBA event service. At present, only one event is defined, the addition of a new LogEntry to the database.

3.16.1.2.3 LogEntry (Class)

This class represents a typical log entry that is stored in the database. This can be a general Communications Log entry or it can be a historical entry for a Traffic Event. Some Traffic Event actions (opening, closing, etc.) are logged in the Communications Log as well as in the history of the specific Traffic Event.

3.16.1.2.4 LogEntryData (Class)

LogEntryData is a collection of data required to create one Log Entry, consisting of text (the body of the event) and an ID that refers to a Traffic Event, if appropriate.

3.16.1.2.5 LogEntryDataList (Class)

The LogEntryDataList is simply a sequence of LogEntryData objects, each of which contain the data needed to create one Log Entry. Normally each LogEntryDataList will contain only one LogEntryData object, but if the CommLog service is unavailable for a time, it is possible that multiple LogEntryData objects may be queued up for insertion into the database.

3.16.1.2.6 LogEntryList (Class)

The LogEntryList is simply a sequence of LogEntry instances returned to a requesting process in one clump. (Some requests return so much data that data is returned in clumps. The initial request returns a LogIterator from which additional LogEntryList sequences can be requested, in order to complete the entire query.

3.16.1.2.7 LogFilter (Class)

This class is used to specify the criteria to be used when getting entries from the Communications Log. The caller would create an object of this type specifying the criteria that each log entry must match in order to be returned.

3.16.1.2.8 LogIterator (Class)

This class represents an iterator to iterate through a collection of log entries. If a retrieval request results in more data than is reasonable to transmit all at once, one clump of entries is returned at first, together with a LogIterator from which additional data can be requested, repeatedly, until all entries are returned or the user cancels the operation.

3.16.1.3 Common (Class Diagram)

This class diagram shows classes used by multiple modules.

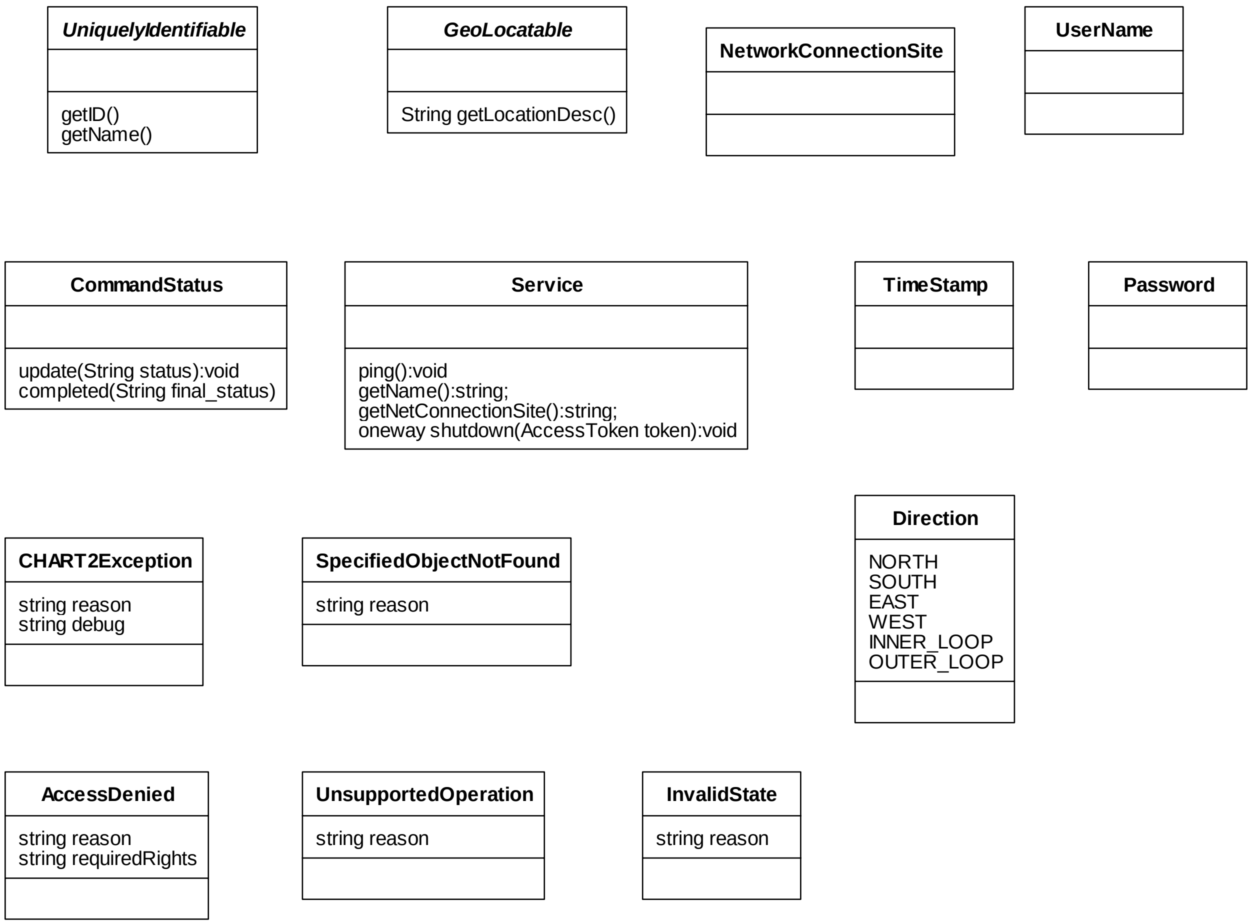

Figure 154. Common (Class Diagram)

3.16.1.3.1 AccessDenied (Class)

This class represents an access denied, or “no rights” failure.

3.16.1.3.2 CHART2Exception (Class)

Generic exception class for the CHART2 system. This class can be used for throwing very generic exceptions that require no special processing by the client. It supports a reason string that may be shown to any user and a debug string that will contain detailed information useful in determining the cause of the problem.

3.16.1.3.3 CommandStatus (Class)

The CommandStatus CORBA interface is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

3.16.1.3.4 Direction (Class)

This enumeration defines direction of travel.

3.16.1.3.5 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.16.1.3.6 InvalidState (Class)

This exception is thrown when an operation is attempted on an object that is not in a valid state to perform the operation.

3.16.1.3.7 NetworkConnectionSite (Class)

The NetworkConnectionSite class contains a string that is used to specify where a service is running. This field is useful for administrators in debugging problems should an object become “software comm failed”. It is included in the CHART2DMSStatus.

3.16.1.3.8 Password (Class)

Typedef used to define the type of a Password.

3.16.1.3.9 Service (Class)

This interface is implemented by all services in the system that allow themselves to be shutdown externally. All implementing classes provide a means to be cleanly shutdown and can be pinged to detect if they are alive.

3.16.1.3.10 SpecifiedObjectNotFound (Class)

Exception used to indicate that an operation was attempted that involves a secondary object that cannot be found by the invoked object.

3.16.1.3.11 TimeStamp (Class)

This typedef defines the type of TimeStamp fields.

3.16.1.3.12 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.16.1.3.13 UnsupportedOperation (Class)

This exception is used to indicate that an operation is not supported by the object on which it is called.

3.16.1.3.14 UserName (Class)

This typedef defines the type of UserName fields used in system interfaces.

3.16.1.4 DeviceManagement (Class Diagram)

This class diagram shows device interfaces that are common among devices.

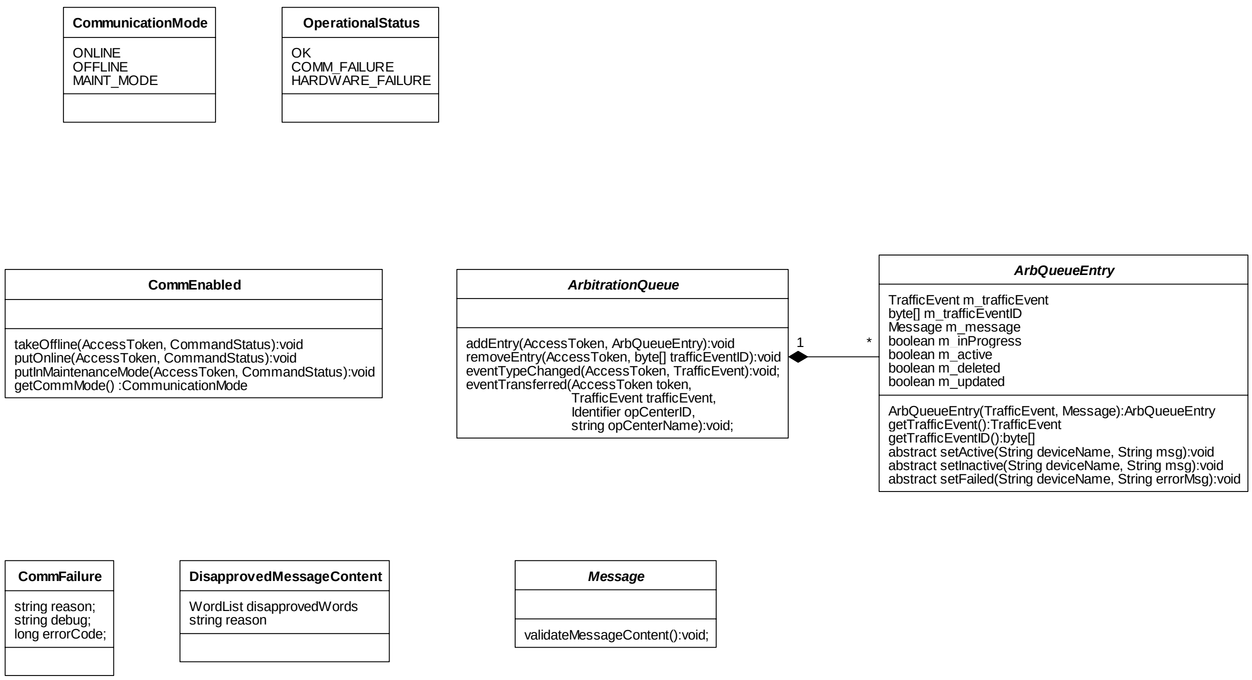

Figure 155. DeviceManagement (Class Diagram)

3.16.1.4.1 ArbitrationQueue (Class)

An ArbitrationQueue is a queue that arbitrates the usage of a device. The arbitration queue determines the proper message to be displayed on a device and switches the active message based on conditions present within the system.

For R1B2, the arbitration queue will have a single slot (and is therefore not really a queue). The device arbitration rules used by the arbitration queue in R1B2 rely on user rights and operations centers. When the arbitration queue's slot is in use, another message can only be added to the queue if the user adding the message is from the same operations center that is responsible for the existing message, or the user has a special functional right that allows this rule to be overridden.

The arbitration queue can be interrupted to keep it from performing its automated processing. This mode is used to allow maintenance on the device being arbitrated by the queue without having the queue's automatic processing interfere with the maintenance activities. While the queue is interrupted, it allows direct commands to be passed to the device for maintenance activies. This feature is built in to allow the device to take advantage of the arbitration queue's asynchronous processing capabilities.

When an interrupted arbitration queue is taken out of its interrupted state through the use of the resume method, the arbitration queue evaluates the messages in the queue and restores the device to the proper state. For R1B2, the arbitration queue simply blanks the device when the queue processing is resumed.

3.16.1.4.2 ArbQueueEntry (Class)

This class is used for an entry on the arbitration queue for a single message for a single traffic event / response plan item. The class holds the associated message, traffic event, and response plan item.

3.16.1.4.3 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.16.1.4.4 CommFailure (Class)

This exception is to be thrown when an error is detected connecting to or communicating with a device.

3.16.1.4.5 CommunicationMode (Class)

The CommunicationMode class enumerations the modes of operation for a DMS: ONLINE, OFFLINE, and MAINT_MODE. The DMSStatus class contains a value of this type.

3.16.1.4.6 DisapprovedMessageContent (Class)

This exception is thrown when a text message to be put on a device contains words that are not approved. This exception is also thrown if an attempt is made to put the device in an invalid display state, such as putting the Beacons ON for a blank DMS.

3.16.1.4.7 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.16.1.4.8 OperationalStatus (Class)

The OperationalStatus class enumerates the types of operational status a DMS can have: OK (normal mode), COMM_FAILURE (no communications to the device), or HARDWARE_FAILURE (device is reachable but is reporting a hardware failure). The DMSStatus class contains a value of this type.

3.16.1.5 DictionaryManagement (Class Diagram)

This class diagram shows the interfaces used for the dictionaries.

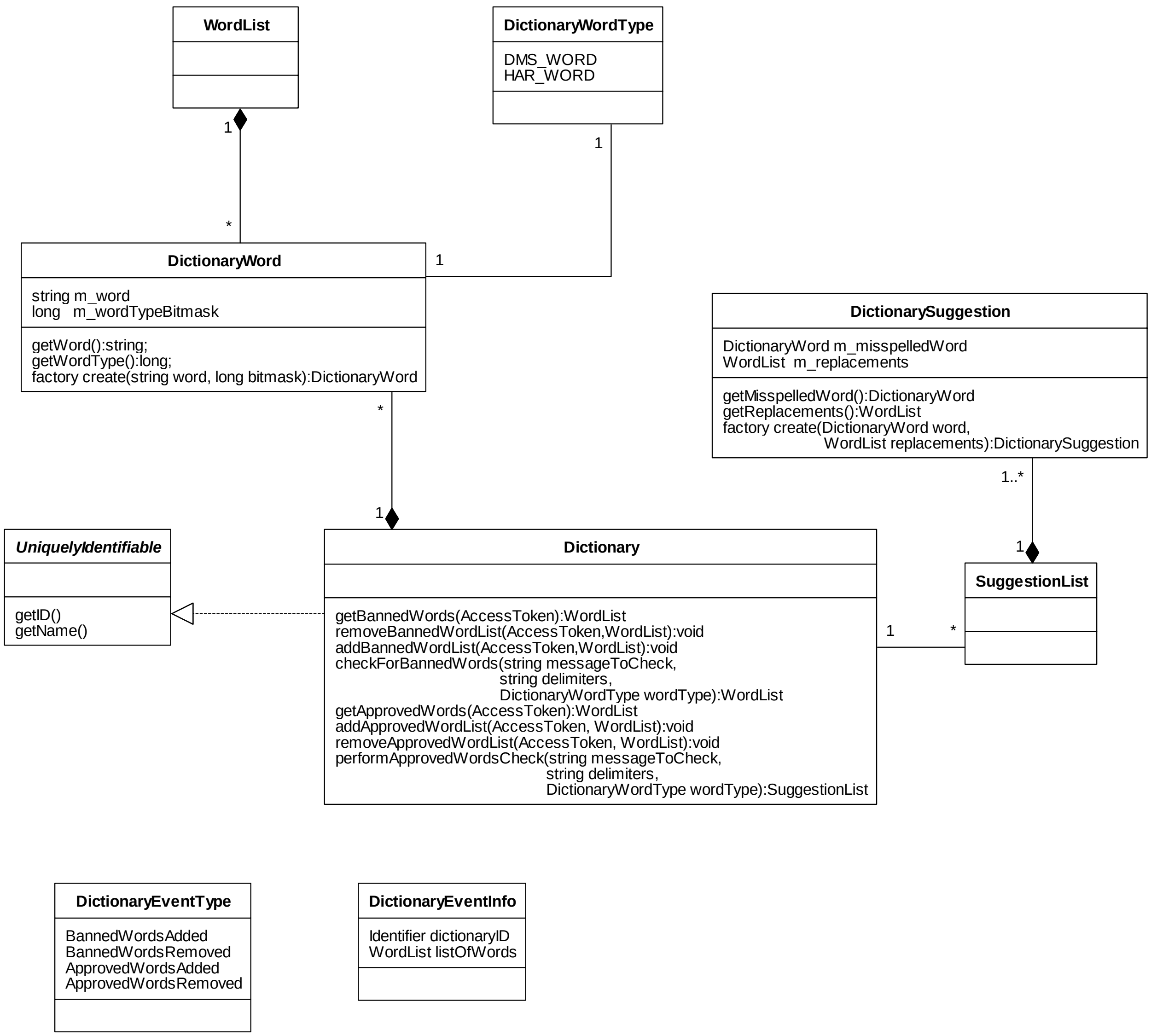

Figure 156. DictionaryManagement (Class Diagram)

3.16.1.5.1 Dictionary (Class)

The Dictionary IDL interface provides functionality to add, delete and check for words that are approved or banned from being used in a CHART2 messaging device. Examples of messaging devices are DMS, HAR etc.

3.16.1.5.2 DictionaryEventInfo (Class)

This interface encapsulates the data that is passed with a dictionary CORBA event. It contains information identifying the dictionary, and the list of words affected by the event.

3.16.1.5.3 DictionaryEventType (Class)

This represents the enumerations used for the different CORBA event types applicable to the dictionary module.

3.16.1.5.4 DictionarySuggestion (Class)

A DictionarySuggestion represents a list of suggested words that may be used as a substitute for the word that could not be found in the approved words dictionary database.

3.16.1.5.5 DictionaryWord (Class)

A DictionaryWord represents a word in the chart2 dictionary. It contains information that qualifies the type of devices that the word applies to.

3.16.1.5.6 DictionaryWordType (Class)

This enumeration is used to tag words that are placed in a dictionary. Words may apply to a specific messaging device or many.

3.16.1.5.7 SuggestionList (Class)

This interface represents the IDL sequence typedef for the DictionarySuggestion.

3.16.1.5.8 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.16.1.5.9 WordList (Class)

This interface represents the IDL sequence typedef for the DictionaryWord.

3.16.1.6 DMSControl (Class Diagram)

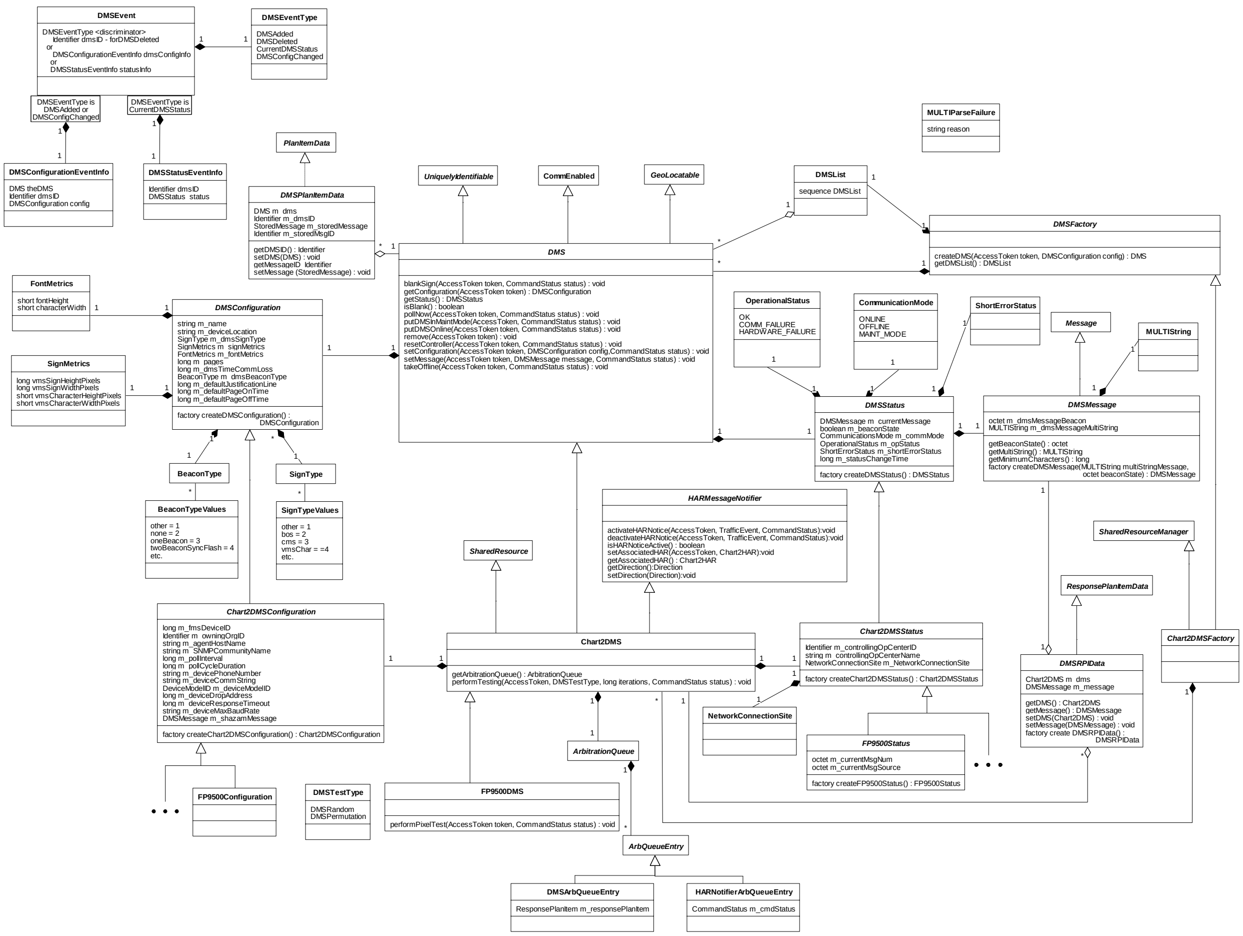

Figure 157. DMSControl (Class Diagram)

3.16.1.6.1 ArbitrationQueue (Class)

An ArbitrationQueue is a queue that arbitrates the usage of a device. The arbitration queue determines the proper message to be displayed on a device and switches the active message based on conditions present within the system.

For R1B2, the arbitration queue will have a single slot (and is therefore not really a queue). The device arbitration rules used by the arbitration queue in R1B2 rely on user rights and operations centers. When the arbitration queue's slot is in use, another message can only be added to the queue if the user adding the message is from the same operations center that is responsible for the existing message, or the user has a special functional right that allows this rule to be overridden.

The arbitration queue can be interrupted to keep it from performing its automated processing. This mode is used to allow maintenance on the device being arbitrated by the queue without having the queue's automatic processing interfere with the maintenance activities. While the queue is interrupted, it allows direct commands to be passed to the device for maintenance activies. This feature is built in to allow the device to take advantage of the arbitration queue's asynchronous processing capabilities.

When an interrupted arbitration queue is taken out of its interrupted state through the use of the resume method, the arbitration queue evaluates the messages in the queue and restores the device to the proper state. For R1B2, the arbitration queue simply blanks the device when the queue processing is resumed.

3.16.1.6.2 ArbQueueEntry (Class)

This class is used for an entry on the arbitration queue for a single message for a single traffic event / response plan item. The class holds the associated message, traffic event, and response plan item.

3.16.1.6.3 BeaconType (Class)

The BeaconType class defines the beacon type for a DMS. Its values are defined by the BeaconTypeValues class. It is a part of a DMSConfiguration object.

3.16.1.6.4 BeaconTypeValues (Class)

The BeaconTypeValues class enumerates the various beacon types used on DMS devices (number of beacons and whether and in what manner they flash).

3.16.1.6.5 CHART2DMS (Class)

The CHART2DMS class extends the DMS interface and defines a more detailed interface to be used in manipulating the CHART II-specific DMS objects within CHART II. It provides a method for getting the DMSArbitrationQueue for a CHART II DMS, which can then be used by traffic events to provide input as to what each traffic event desires to be on the sign. It also provides a method to perform testing on a sign. This method can be extended by derived classes for specific models of signs, which know how to perform certain types of testing on their specific model of sign. CHART II business rules include concepts such as shared resrouces, arbitration queues, and linking devices usage to traffic events, concepts which go beyond what would be industry-standard DMS control.

3.16.1.6.6 CHART2DMSConfiguration (Class)

The CHART2DMSConfiguration class is an abstract class which extends the DMSConfiguration class to provide configuration information specific to CHART II processing. Such information includes how to contact the sign under CHART II software control, the default SHAZAM message for using the sign as a HAR Notifier, and the owning organization. Such data extends beyond what would be industry-standard configuration information for a DMS.

3.16.1.6.7 CHART2DMSFactory (Class)

The CHART2DMSFactory class extends the DMSFactory interface to provide additional CHART II specific capability. This factory creates CHART2DMS objects (extensions of DMS objects). It implements SharedResourceManager capbility control DMS objects as shared resources.

3.16.1.6.8 CHART2DMSStatus (Class)

The CHART2DMSStatus class is an abstract class that extends the DMSStatus class to provide status information specific to CHART II processing, such as information on the controlling operations center for the sign. This data extends beyond what would be industry-standard status information for a DMS.

3.16.1.6.9 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.16.1.6.10 CommunicationMode (Class)

The CommunicationMode class enumerations the modes of operation for a DMS: ONLINE, OFFLINE, and MAINT_MODE. The DMSStatus class contains a value of this type.

3.16.1.6.11 DMS (Class)

The DMS class defines an interface to be used in manipulating Dynamic Message Sign (DMS) objects within CHART II. It specifies methods for setting messages and clearing messages from a sign (in maintenance mode), polling a sign, changing the configuration of a sign, and reseting a sign. (Setting messages on a sign in online mode are not accomplished by manipulating a DMS directly; that is accomplished by manipulating traffic events, which interfaces with the DMSArbitrationQueue of a sign. This activity involves the DMS extension, CHART2DMS, which defines interactions with signs under CHART II business rules.)

3.16.1.6.12 DMSArbQueueEntry (Class)

The DMSArbQueueEntry class provides an implementation of ArbQueueEntry that is used for most standard entries placed on the arbitration queue. When its setActive, setInactive, and setFailed methods are called, it adds a log entry to its traffic event and calls the appropriate method on its response plan item (setActive, setInactive, or update).

3.16.1.6.13 DMSConfiguration (Class)

The DMSConfiguration class is an abstract class that describes the configuration of a DMS device. This configuration information is normally fairly static: things like the size of the sign in characters and pixels, its name and location, and how to contact the sign (as opposed to dynamic information like the current message on the sign, which is defined in an analogous Status object).

3.16.1.6.14 DMSConfigurationEventInfo (Class)

The DMSConfigurationEventInfo class is the type of DMSEvent used for DMSEventType DMSConfigChanged. It contains a DMSConfiguration object that details the new configuration for a CHART II DMS object.

3.16.1.6.15 DMSEvent (Class)

The DMSEvent class is a union which can be any one of four events relating to DMS operations which can be pushed on an Event Channel to update event consumers on DMS-related activities. The four types of events, defined by the enumeration DMSEventType, are: DMSAdded, DMSDeleted, CurrentDMSStatus, and DMSConfigChanged.

3.16.1.6.16 DMSEventType (Class)

The DMSEventType is an enumeration which defines the four types of events relating to DMS operations which can be pushed on an Event Channel to update event consumers on DMS-related activities. The four types of events are: DMSAdded, DMSDeleted, CurrentDMSStatus, and DMSConfigChanged.

3.16.1.6.17 DMSFactory (Class)

The DMSFactory class specifies the interface to be used to create DMS objects within the CHART II system. It also provides a method to get a list of DMS devices currently in the system.

3.16.1.6.18 DMSList (Class)

The DMSList class is simply a list of DMS devices which can be used by the DMS Factory and other classes for maintaining the list or other lists of DMS objects.

3.16.1.6.19 DMSMessage (Class)

The DMSMessage class is an abstract class that describes a message for a DMS. It consists of two elements: a MULTI-formatted message and beacon state information (whether the message requires that the beacons be on). The DMSMessage is contained within a DMSStatus object, used to communicate the current message on a sign, and so within a DMSRPIData object, used to specify the message that should be on a sign when the response plan item is executed.

3.16.1.6.20 DMSPlanItemData (Class)

The DMSPlanItemData class is a valuetype that contains data stored in a plan item for a DMS. It is derived from PlanItemData.

3.16.1.6.21 DMSRPIData (Class)

The DMSRPIData class is an abstract class that describes a response plan item for a DMS. It contains the unique identifier of the DMS to contain the DMSMessage, and the DMSMessage itself.

3.16.1.6.22 DMSStatus (Class)

The DMSStatus class is an abstract value-type class that provides status information for a DMS. This status information is relatively dynamic: things like the current message on the sign, its beacon state, its current operational mode (online, offline, maintenance mode), and current operational status (OK, COMM_FAILURE, or HARDWARE_FAILURE). (More static information about the sign, such as its size and location, is defined in an analogous Configuration object.)

3.16.1.6.23 DMSStatusEventInfo (Class)

The DMSStatusEventInfo class is the type of DMSEvent used for DMSEventType CurrentDMSStatus. It contains a DMSStatus object that details the new status for a CHART II DMS object.

3.16.1.6.24 DMSTestType (Class)

The DMSTestType enumeration identifies two types of tests which can be performed on DMS devices: random and permutation.

3.16.1.6.25 FontMetrics (Class)

The FontMetrics class is a non-behavioral class (structure) which contains information regarding to the font size used on a DMS. It is a part of a DMSConfiguration object.

3.16.1.6.26 FP9500Configuration (Class)

The FP9500Configuration class is an abstract class that extends the CHART2DMSConfiguration class to provide configuration information specific to an FP9500 model of DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information.

3.16.1.6.27 FP9500DMS (Class)

The FP9500DMS class extends the CHART2DMS interface and defines a more detailed interface to be used in manipulating FP9500 models of DMS signs. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific DMS control. For instance, the FP9500DMS has a performPixelTest method, which knows how to invoke and interpret a pixel test as supported by the FP9500 model DMS.

3.16.1.6.28 FP9500Status (Class)

The FP9500Status class is an abstract class that extends the CHART2DMSStatus class to provide status information specific to an FP9500 model of DMS. It is exemplary of potentially a whole suite of subclasses specific to a specific brand and model of sign for manufacturer-specific configuration information. In this case, additional information provided the FP9500 model includes the current message number and current message source.

3.16.1.6.29 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.16.1.6.30 HARMessageNotifier (Class)

The HARMessageNotifier class specifies an interface to be implemented by devices that can be used to notify the traveler to tune in to a radio station to hear a traffic message being broadcast by a HAR. A HARMessageNotifier is directional and allows users of the device to better determine if activation of the device is warranted for the message being broadcast by the HAR. This interface can be implemented by SHAZAMs and by DMS devices that are allowed to provide a SHAZAM-like message in the absence of any more useful messages to display.

3.16.1.6.31 HARNotifierArbQueueEntry (Class)

The HarNotifierArbQueueEntry class provides an implementation of the ArbQueueEntry used for entries that are placed on the arbitration queue to put a “SHAZAM” message on a DMS. These types of messages have a low priority and are not allowed to overwrite any standard message (from a DMSArbQueueEntry) that is currently displayed on a device. These types of messages are also different in that they are not added to the queue directly by a response plan item and are instead included as a sub-task of activating a message on a HAR. The HAR uses a command status object to track the progress of the HAR notifier message.

3.16.1.6.32 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.16.1.6.33 MULTIParseFailure (Class)

The MULTIParseFailure class is an exception to be thrown when a MULTI-formatted DMS message cannot be correctly parsed.

3.16.1.6.34 MULTIString (Class)

The MULTIString class is a MULTI-formatted DMS message. The DMSMessage class contains a MULTIString value to specify the content of the sign, in addition to the beacon state value.

3.16.1.6.35 NetworkConnectionSite (Class)

The NetworkConnectionSite class contains a string that is used to specify where a service is running. This field is useful for administrators in debugging problems should an object become “software comm failed”. It is included in the CHART2DMSStatus.

3.16.1.6.36 OperationalStatus (Class)

The OperationalStatus class enumerates the types of operational status a DMS can have: OK (normal mode), COMM_FAILURE (no communications to the device), or HARDWARE_FAILURE (device is reachable but is reporting a hardware failure). The DMSStatus class contains a value of this type.

3.16.1.6.37 PlanItemData (Class)

This class is a valuetype that is the base class for data stored in a plan item. Derived classes contain specific data that map a device to an operation and the data needed for the operation. For example a derived class provides a mapping between a specific DMS and a DMSMessage.

3.16.1.6.38 ResponsePlanItemData (Class)

This class is a delegate used to perform the execute and remove tasks for the response plan item. Derived classes of this base class have specific implementations for the type of device the response plan item is used to control.

3.16.1.6.39 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.16.1.6.40 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn't have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.16.1.6.41 ShortErrorStatus (Class)

The ShortErrorStatus class identifies an error condition for a DMS. It is a bit field defined by the NTCIP center to field standard for DMS that specifies error conditions that may be present on the device. This class is used to encapsulate the bit mask and provide a user-friendly interface to the error conditions. The DMSStatus class contains a value of this type.

3.16.1.6.42 SignMetrics (Class)

The SignMetrics class is a non-behavioral class (structure) which contains information regarding to the size of a DMS, in pixels and characters. It is a part of a DMSConfiguration object.

3.16.1.6.43 SignType (Class)

The SignType class defines the sign type for a DMS. Its values are defined by the SignTypeValues class. It is a part of a DMSConfiguration object.

3.16.1.6.44 SignTypeValues (Class)

The SignTypeValues class enumerates the various sign types DMS devices. Examples are bos, cms, vmsChar, etc.

3.16.1.6.45 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.16.1.7 PlanManagement (Class Diagram)

This class diagram contains the interfaces used in the creation and management of plans. A plan is a group of actions that are set-up in advance to be used in response to a traffic event. Given the unpredictable nature of traffic events, pre-defined plans are usually only useful for congestion, safety messages, and weather-related messages.

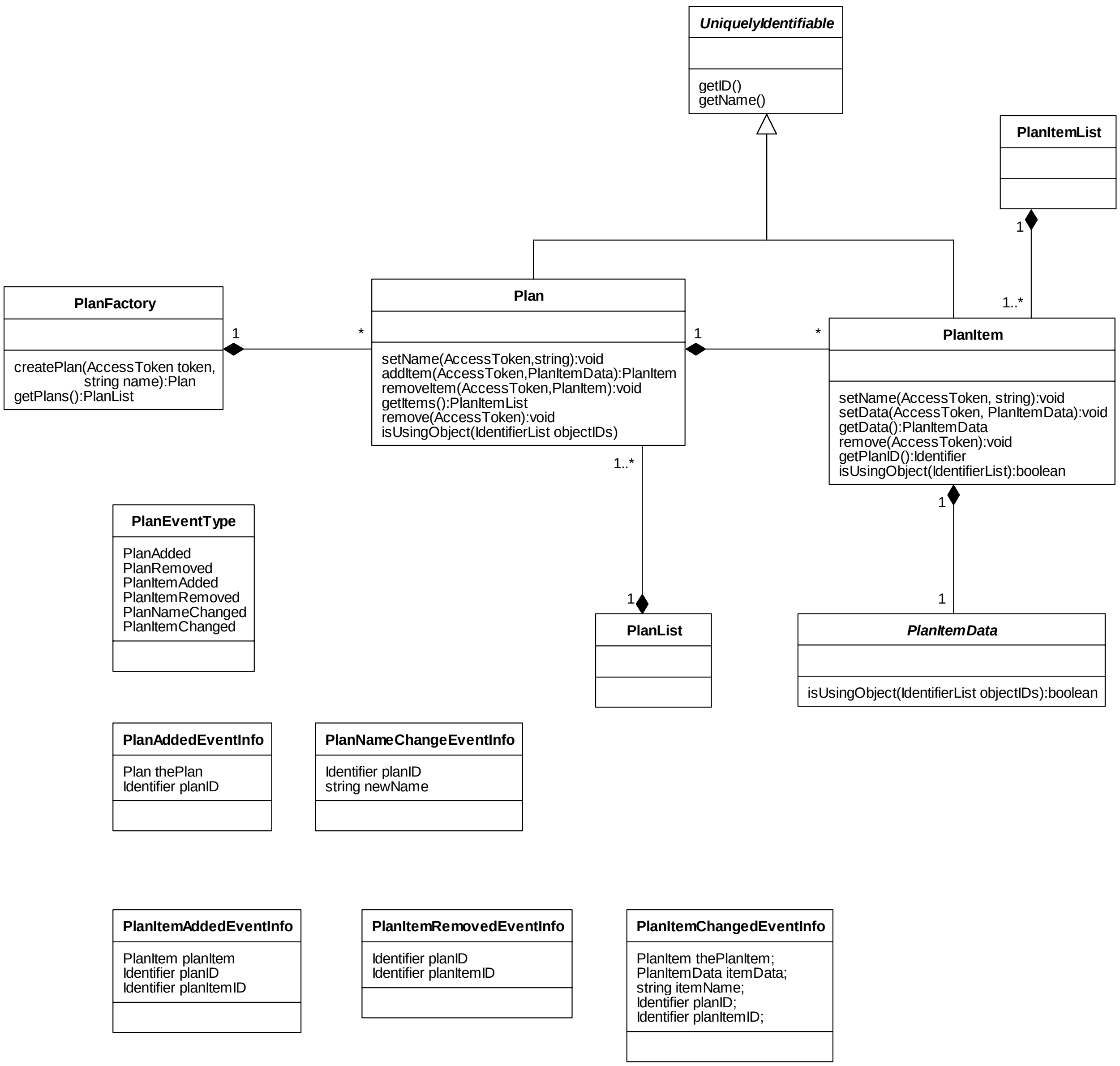

Figure 158. PlanManagement (Class Diagram)

3.16.1.7.1 Plan (Class)

A Plan is a group of actions listed out in advance to be used in response to a traffic event. Each action is defined to be a Plan item. The Plan supports functionality to add and remove plan items.

3.16.1.7.2 PlanAddedEventInfo (Class)

The PlanAddedEventInfo class defines the data passed in the PlanAdded event.

3.16.1.7.3 PlanEventType (Class)

The PlanEventType class is an enumeration that describes the types of events that can be pushed for plans. When a plan item is added or modified it is up to the derived item type to push the appropriate type of event.

3.16.1.7.4 PlanFactory (Class)

This class creates, destroys, and maintains the collection of plans that can be used in the system.

3.16.1.7.5 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This CORBA interface is subclassed for specific actions that can be planned in the system.

3.16.1.7.6 PlanItemAddedEventInfo (Class)

The PlanItemAddededEventInfo class defines the data passed in the PlanItemAdded event.

3.16.1.7.7 PlanItemChangedEventInfo (Class)

The PlanItemChangedEventInfo class defines the data passed in the PlanItemChanged event.

3.16.1.7.8 PlanItemData (Class)

This class is a valuetype that is the base class for data stored in a plan item. Derived classes contain specific data that map a device to an operation and the data needed for the operation. For example a derived class provides a mapping between a specific DMS and a DMSMessage.

3.16.1.7.9 PlanItemList (Class)

The PlanItemList class is simply a collection of PlanItem objects.

3.16.1.7.10 PlanItemRemovedEventInfo (Class)

The PlanItemRemovedEventInfo defines the data passed in the PlanItemRemoved event.

3.16.1.7.11 PlanList (Class)

The PlanList class is simply a collection of Plan objects.

3.16.1.7.12 PlanNameChangeEventInfo (Class)

The PlanNameChangeEventInfo class defines the data passed in the PlanNameChanged event.

3.16.1.7.13 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.16.1.8 HARControl (Class Diagram)

This class diagram contains the interfaces relating to the control of Highway Advisory Radio (HAR).

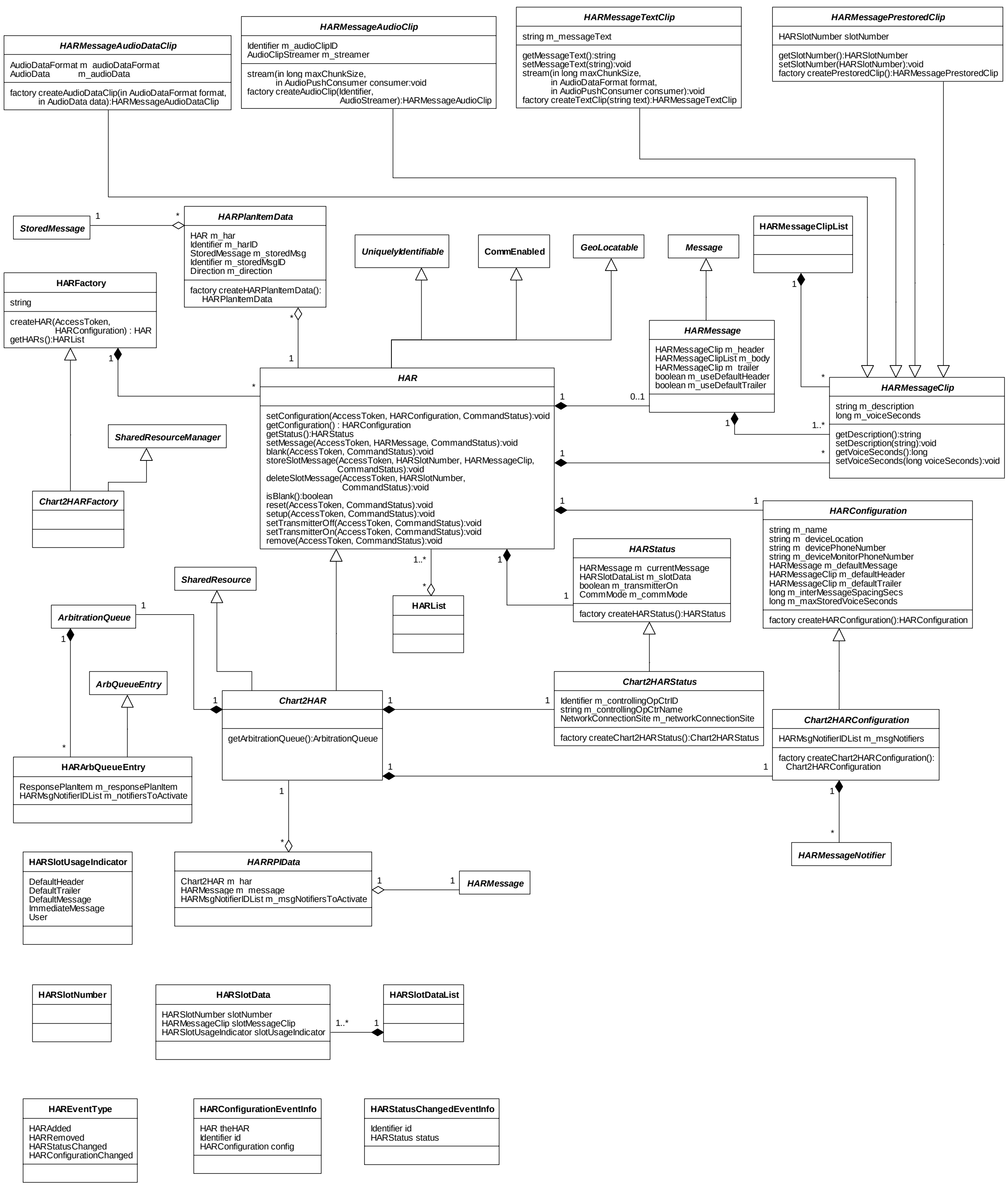

Figure 159. HARControl (Class Diagram)

3.16.1.8.1 ArbitrationQueue (Class)

An ArbitrationQueue is a queue that arbitrates the usage of a device. The arbitration queue determines the proper message to be displayed on a device and switches the active message based on conditions present within the system.

For R1B2, the arbitration queue will have a single slot (and is therefore not really a queue). The device arbitration rules used by the arbitration queue in R1B2 rely on user rights and operations centers. When the arbitration queue's slot is in use, another message can only be added to the queue if the user adding the message is from the same operations center—the one responsible for the existing message—or the user has a special functional right that allows this rule to be overridden.

The arbitration queue can be interrupted to keep it from performing its automated processing. This mode is used to allow maintenance on the device being arbitrated by the queue without having the queue's automatic processing interfere with the maintenance activities.

When an interrupted arbitration queue is taken out of its interrupted state through the use of the resume method, the arbitration queue evaluates the messages in the queue and restores the device to the proper state. For R1B2, the arbitration queue performs no special processing when resumed because the queue cleans itself when interrupted and does not allow new entries while interrupted.

3.16.1.8.2 ArbQueueEntry (Class)

This class is used for an entry on the arbitration queue for a single message for a single traffic event / response plan item. The class holds the associated message, traffic event, and response plan item.

3.16.1.8.3 CHART2HAR (Class)

The CHART2HAR class is an extension of the HAR that is aware of CHART2 business rules, such as arbitration queues, linking device usage to traffic events, and the concept of a shared resource.

3.16.1.8.4 CHART2HARConfiguration (Class)

This class contains configuration data for the HAR that is used for CHART II specific processing (as opposed to the configuration values contained in HARConfiguration that relate to typical HAR usage).

3.16.1.8.5 CHART2HARFactory (Class)

This interface defines objects capable of creating CHART2HAR objects. This factory is also responsible for monitoring the HARs as shared resources and must report when a HAR that is currently broadcasting a message (other than the default) does not have a user logged into the system that is from the controlling operations center.

3.16.1.8.6 CHART2HARStatus (Class)

This class contains status information for a CHART2HAR object. This information is specific to CHART II processing and extends beyond the status related to typical HAR device control.

3.16.1.8.7 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.16.1.8.8 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.16.1.8.9 HAR (Class)

This class is used to represent a Highway Advisory Radio (HAR) device. A HAR is used to broadcast traffic related information over a localized radio transmitter, making the information available to the traveler.

3.16.1.8.10 HARArbQueueEntry (Class)

This class is an arbitration queue entry used to set the message on a HAR on behalf of a traffic event. This entry also specifies the HARMessageNotifiers to be activated when the message is activated.

3.16.1.8.11 HARConfiguration (Class)

This class contains configuration data for a HAR device.

3.16.1.8.12 HARConfigurationEventInfo (Class)

This class defines data pushed with a HARConfigurationChanged and HARAdded CORBA event.

3.16.1.8.13 HAREventType (Class)

This enumeration defines the types of CORBA events that are pushed on a HARControl event channel.

3.16.1.8.14 HARFactory (Class)

This CORBA interface allows new HAR objects to be added to the system.

3.16.1.8.15 HARList (Class)

The HARList class is simply a collection of HAR objects.

3.16.1.8.16 HARMessage (Class)

This utility class represents a message that is capable of being stored on a HAR. It stores the HAR message as a HAR message header, body and footer. It contains methods to input and output them in different formats.

3.16.1.8.17 HARMessageAudioClip (Class)

This class is a thin wrapper for recorded voice that is to be played on a HAR. This class is passed around the system instead of passing the actual voice data. When the actual voice data is needed to play to the user or to program the HAR device, this object's streamer is used to stream the actual voice data.

3.16.1.8.18 HARMessageAudioDataClip (Class)

This class is a message clip that contains audio data and the format of the audio data. Because audio data can be very large, this type of clip is reserved for use when recorded voice is first entered into the system. Recorded voice that already exists in the system is passed throughout the system using HARMessageAudioClip to avoid sending the large audio data when possible.

3.16.1.8.19 HARMessageClip (Class)

This class represents a section of a HAR message. It can be either plain text that would need to be converted to audio prior to broadcast, or binary format (MP3, WAV, etc.)

3.16.1.8.20 HARMessageClipList (Class)

The HARMessageClipList is a collection of HARMessageClip objects.

3.16.1.8.21 HARMessageNotifier (Class)

The HARMessageNotifier class specifies an interface to be implemented by devices that can be used to notify the traveler to tune in to a radio station to hear a traffic message being broadcast by a HAR. A HARMessageNotifier is directional and allows users of the device to better determine if activation of the device is warranted for the message being broadcast by the HAR. This interface can be implemented by SHAZAMs and by DMS devices that are allowed to provide a SHAZAM-like message in the absence of any more useful messages to display.

3.16.1.8.22 HARMessagePrestoredClip (Class)

This class stores data used to identify the usage of a clip that has already been stored on a HAR device.

3.16.1.8.23 HARMessageTextClip (Class)

This class represents a HAR message content object that is in plain text format. This message can be checked for banned words and will be converted into a voice message using a speech engine to broadcast on a HAR device.

3.16.1.8.24 HARPlanItemData (Class)

This class is used to associate a message with a HAR for use in Plans.

3.16.1.8.25 HARRPIData (Class)

This class represents an item in a traffic event response plan that is capable of issuing a command to put a message on a HAR when executed. When the item is executed, it adds the message to the arbitration queue of the specified HAR. When the item is removed from the response plan (manually or implicitly through closing the traffic event) the item asks the HAR's arbitration queue to remove the message.

3.16.1.8.26 HARSlotData (Class)

This struct defines the data used to identify the contents of a slot in the HAR controller.

3.16.1.8.27 HARSlotDataList (Class)

The HARSlotDataList class is simply a collection of HARSlotData objects.

3.16.1.8.28 HARSlotNumber (Class)

The HARSlotNumber is an integer used to specify slot numbers on a HAR controller.

3.16.1.8.29 HARSlotUsageIndicator (Class)

This enum defines indicators used to show the usage of a given slot in the HAR controller.

3.16.1.8.30 HARStatus (Class)

This class contains data that indicates the current status of a HAR device.

3.16.1.8.31 HARStatusChangedEventInfo (Class)

This class contains data that is pushed when the HARStatusChanged CORBA event is pushed on the HARControl event channel.

3.16.1.8.32 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.16.1.8.33 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.16.1.8.34 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn't have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.16.1.8.35 StoredMessage (Class)

This class holds a message object that is stored in a message in a library. It contains attributes such as category and message description, which are used to allow the user to organize messages.

3.16.1.8.36 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.16.1.9 ResourceManagement (Class Diagram)

This class diagram contains the interfaces pertaining to shared resources, operations centers, user login sessions, and organizations.

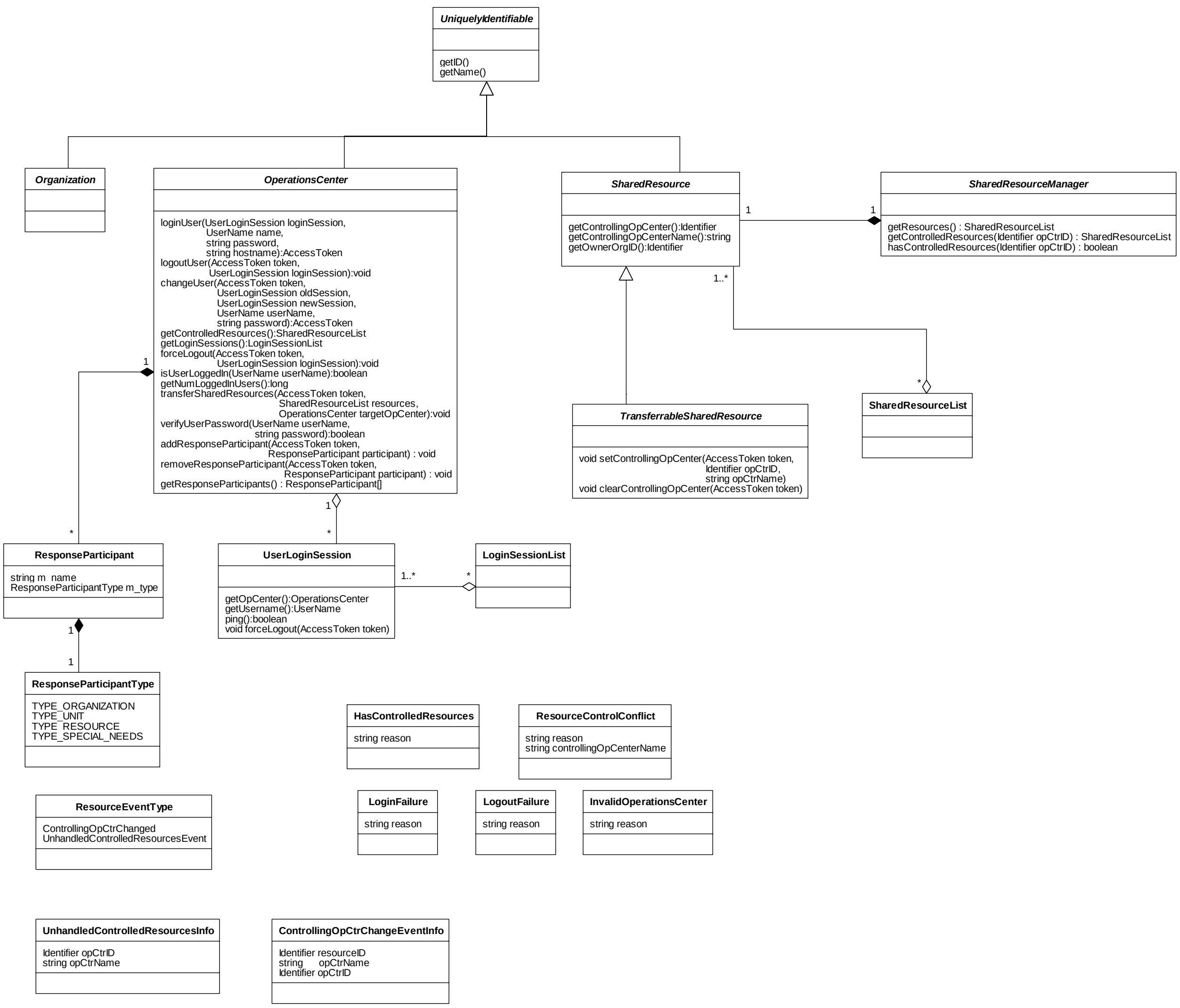

Figure 160. ResourceManagement (Class Diagram)

3.16.1.9.1 ControllingOpCtrChangeEventInfo (Class)

The ControllingOpCtrChangeEventInfo class defines data to be passed on a ControllingOpCtrChange event.

3.16.1.9.2 HasControlledResources (Class)

This class represents an exception which describes a failure caused when the user tries to do something which requires that no resources be controlled, yet the Operations Center which the user is logged in to is still controlling one or more shared resources.

3.16.1.9.3 InvalidOperationsCenter (Class)

Exception that describes a failure caused when the operations center specified is not valid for the attempted operation.

3.16.1.9.4 LoginFailure (Class)

This class represents an exception that describes a login failure.

3.16.1.9.5 LoginSessionList (Class)

A LoginSessionList is simply a collection of UserLoginSession objects.

3.16.1.9.6 LogoutFailure (Class)

This exception is thrown when an error occurs while logging a user out of the system.

3.16.1.9.7 OperationsCenter (Class)

The OperationsCenter represents a center where one or more users are located. This class is used to log users into the system. If the username and password provided to the loginUser method are valid, the caller is given a token that contains information about the user and the functional rights of the user. This token is then used to call privileged methods within the system. Shared resources in the system are either available or under the control of an OperationsCenter. The OperationsCenter keeps track of users that are logged in so that it can ensure that the last user does not log out while there are shared resources under its control. This list of logged in users is also available for monitoring system usage or to force users to logout for system maintenance.

3.16.1.9.8 Organization (Class)

The Organization interface extends the UniquelyIdentifiable interface and will represent an organization, that is an administrative body that can control or own resources.

3.16.1.9.9 ResourceControlConflict (Class)

This exception is thrown when attempt to gain control of a shared resource fails because the resource is under the control of a different operations center and the requesting user does not have the functional right to override the restriction.

3.16.1.9.10 ResourceEventType (Class)

The ResourceEventType enumeration defines all of the resource related event types.

3.16.1.9.11 ResponseParticipant (Class)

The ResponseParticipant class is a non-behavioral structure that specifies a participant in a response.

3.16.1.9.12 ResponseParticipantType (Class)

The ResponseParticipantType enumeration defines a type of entity participating in a response to an event. This could be an external organization, a mobile unit, a mobile device or special purpose vehicle, or a special needs vehicle equipped to handle unusual or hazardous situations.

3.16.1.9.13 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.16.1.9.14 SharedResourceList (Class)

A SharedResourceList is simply a collection of SharedResource objects.

3.16.1.9.15 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn’t have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.16.1.9.16 TransferrableSharedResource (Class)

The TransferrableSharedResource interface extends the SharedResource interface, which is implemented by SharedResource objects whose control can be transferred from one operations center to another.

3.16.1.9.17 UnhandledControlledResourcesInfo (Class)

The UnhandledControlledResourcesEvent class is an event pushed when it is detected that an OperationsCenter is controlling one or more controlled resources but has no users logged in.

3.16.1.9.18 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.16.1.9.19 UserLoginSession (Class)

The UserLoginSession CORBA interface is used to store information about a user that is logged into the system. This object is served from the GUI and provides a means for the servers to call back into the GUI process.

3.16.1.10 HARNotification (Class Diagram)

This Class Diagram shows the classes involved in manipulating HAR message notifications. The HAR notifiers can be SHAZAMs or DMS devices that are acting as SHAZAMs. Note that R1B2 prevents a DMS SHAZAM message from overwriting another type of DMS message.

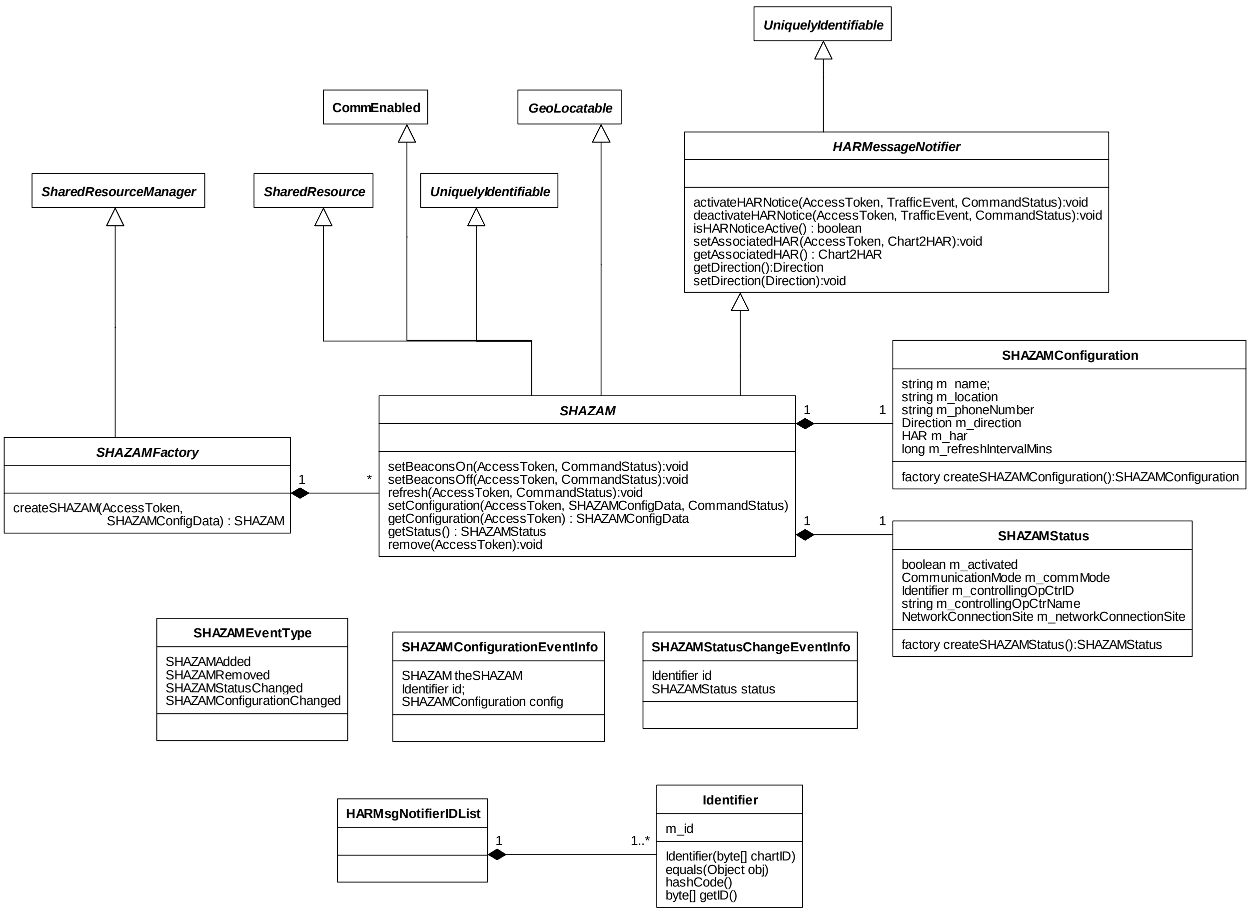

Figure 161. HARNotification (Class Diagram)

3.16.1.10.1 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can be taken offline, put online, or put in maintenance mode. These states typically apply only to field devices. When a device is taken offline, it is no longer available for use through the system and automated polling is halted. When put online, a device is again available for use through the system and automated polling is enbled (if applicable). When put in maintenance mode a device is offline except that maintenance commands to the device are allowed to help in troubleshooting.

3.16.1.10.2 GeoLocatable (Class)

This interface is implemented by objects that can provide location information to their users.

3.16.1.10.3 HARMessageNotifier (Class)

The HARMessageNotifier class specifies an interface to be implemented by devices that can be used to notify the traveler to tune in to a radio station to hear a traffic message being broadcast by a HAR. A HARMessageNotifier is directional and allows users of the device to better determine if activation of the device is warranted for the message being broadcast by the HAR. This interface can be implemented by SHAZAMs and by DMS devices that are allowed to provide a SHAZAM-like message in the absence of any more useful messages to display.

3.16.1.10.4 HARMsgNotifierIDList (Class)

This typedef is a sequence of HARMessageNotifier identifiers.

3.16.1.10.5 Identifier (Class)

Wrapper class for a CHART2 identifier byte sequence. This class will be used to add identifiable objects to hash tables and perform subsequent lookup operations.

3.16.1.10.6 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

3.16.1.10.7 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn’t have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.16.1.10.8 SHAZAM (Class)

This class is used to represent a SHAZAM field device. This class uses a helper class to perform the model specific protocol for device command and control.

3.16.1.10.9 SHAZAMConfiguration (Class)

This class contains data that specifies the configuration of a SHAZAM device.

3.16.1.10.10 SHAZAMConfigurationEventInfo (Class)

This class contains data that is pushed on the SHAZAMControl CORBA event channel with a SHAZAMConfigurationChanged or SHAZAMAdded event type.

3.16.1.10.11 SHAZAMEventType (Class)

This enum defines the types of CORBA events that are pushed on a SHAZAM control event channel.

3.16.1.10.12 SHAZAMFactory (Class)

This CORBA interface allows new SHAZAM objects to be added to the system.

3.16.1.10.13 SHAZAMStatus (Class)

This class contains the current status of a SHAZAM device.

3.16.1.10.14 SHAZAMStatusChangeEventInfo (Class)

This class contains data that is pushed on a SHAZAMControl event channel with a SHAZAMStatusChanged event.

3.16.1.10.15 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.16.1.11 LibraryManagement (Class Diagram)

This class diagram shows all classes and relationships relating to message libaries.

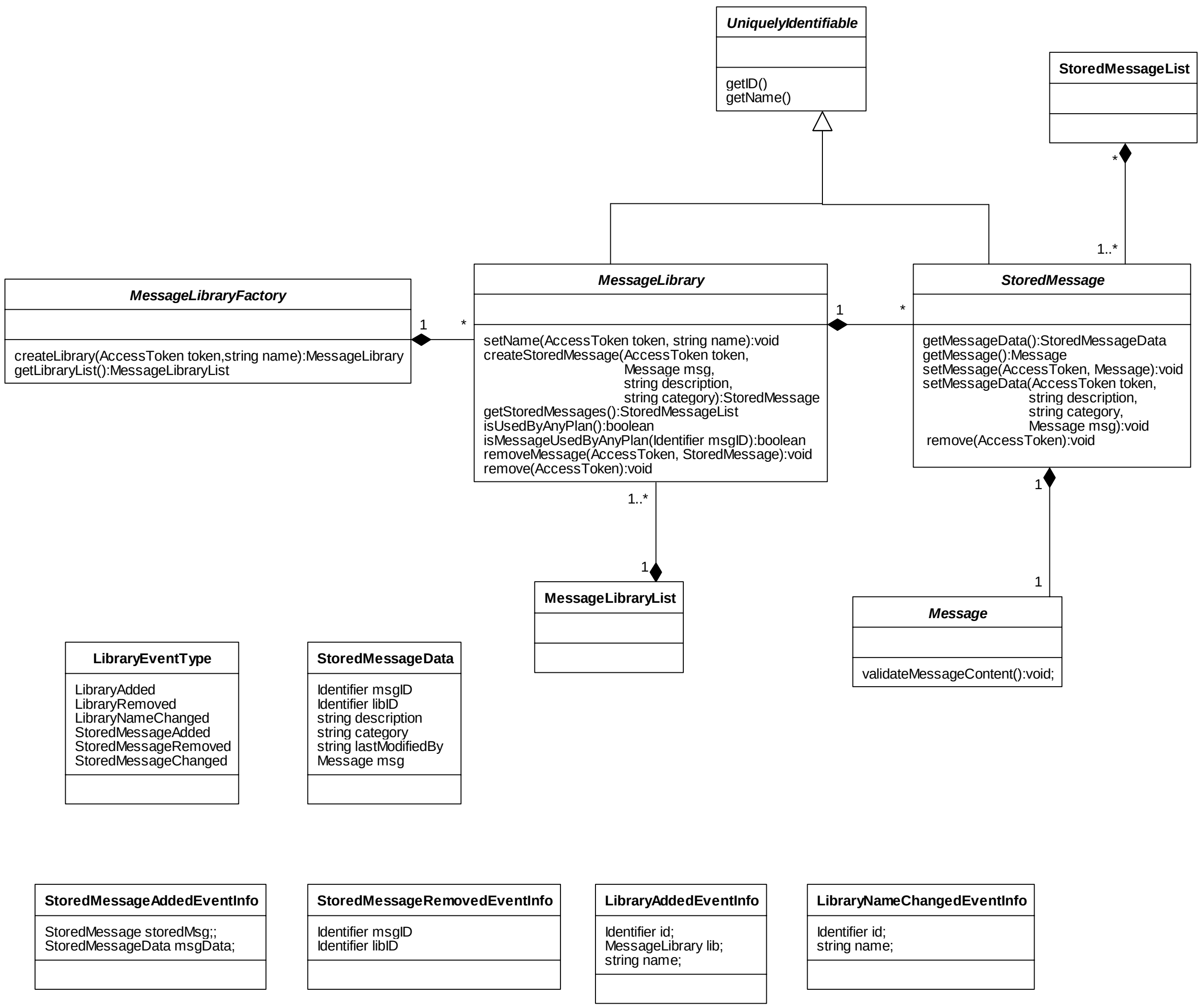

Figure 162. LibraryManagement (Class Diagram)

3.16.1.11.1 LibraryAddedEventInfo (Class)

This struct defines data passed with a DMSLibraryAdded event.

3.16.1.11.2 LibraryEventType (Class)

This enum defines the types of events that can be pushed on a LibraryManagement event channel.

3.16.1.11.3 LibraryNameChangedEventInfo (Class)

This struct defines data passed with a LibraryNameChanged event.

3.16.1.11.4 Message (Class)

This class represents a message that will be used while activating devices. This class provides a means to check if the message contains any banned words given a Dictionary object. Derived classes extend this class to provide device specific message data.

3.16.1.11.5 MessageLibrary (Class)

This class represents a logical collection of messages that are stored in the database.

3.16.1.11.6 MessageLibraryFactory (Class)

This class is used to create new message libraries and maintain them in a collection.

3.16.1.11.7 MessageLibraryList (Class)

A collection of MessageLibrary objects.

3.16.1.11.8 StoredMessage (Class)

This class holds a message object that is stored in a message in a library. It contains attributes such as category and message description which are used to allow the user to organize messages.

3.16.1.11.9 StoredMessageAddedEventInfo (Class)

This struct defines the data passed with a StoredMessageAdded event.

3.16.1.11.10 StoredMessageData (Class)

This structure defines the data stored in a StoredMessage.

3.16.1.11.11 StoredMessageList (Class)

A collection of StoredMessage objects.

3.16.1.11.12 StoredMessageRemovedEventInfo (Class)

This struct defines data passed with a StoredMessageRemoved event.

3.16.1.11.13 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.16.1.12 LogCommon (Class Diagram)

This class diagram contains all interfaces that are necessary to multiple log types within the CHART II system.

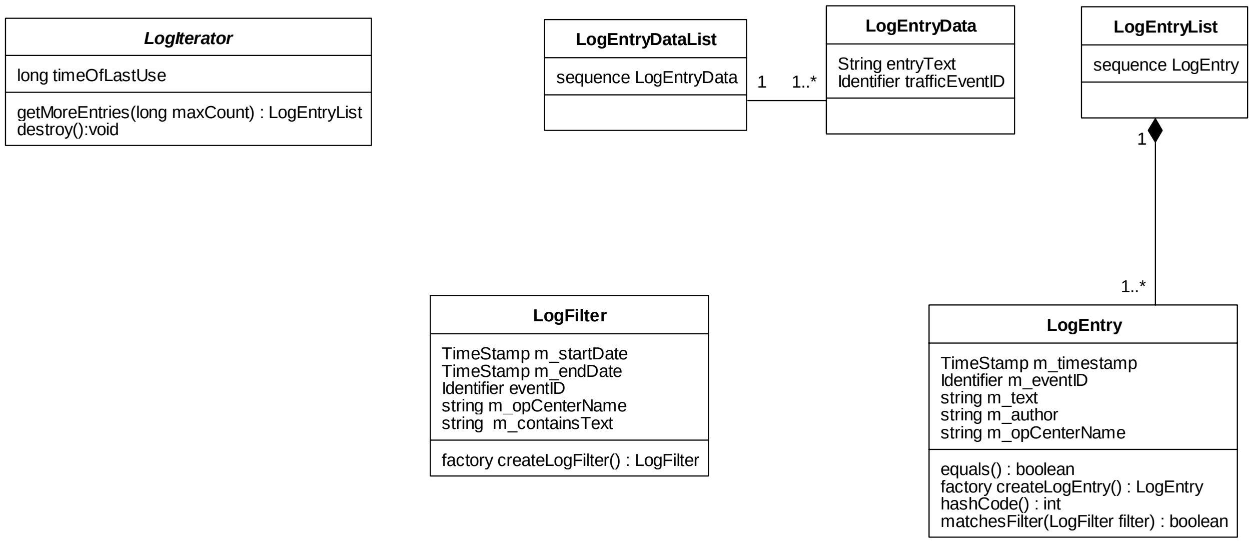

Figure 163. LogCommon (Class Diagram)

3.16.1.12.1 LogEntry (Class)

This class represents a typical log entry that is stored in the database. This can be a general Communications Log entry or it can be a historical entry for a Traffic Event. Some Traffic Event actions (opening, closing, etc.) are logged in the Communications Log as well as in the history of the specific Traffic Event.

3.16.1.12.2 LogEntryData (Class)

LogEntryData is a collection of data required to create one Log Entry, consisting of text (the body of the event) and an ID that refers to a Traffic Event, if appropriate.

3.16.1.12.3 LogEntryDataList (Class)

The LogEntryDataList is simply a sequence of LogEntryData objects, each of which contain the data needed to create one Log Entry. Normally each LogEntryDataList will contain only one LogEntryData object, but if the CommLog service is unavailable for a time, it is possible that multiple LogEntryData objects may be queued up for insertion into the database.

3.16.1.12.4 LogEntryList (Class)

The LogEntryList is simply a sequence of LogEntry instances returned to a requesting process in one clump. (Some requests return so much data that data is returned in clumps. The initial request returns a LogIterator from which additional LogEntryList sequences can be requested, in order to complete the entire query.

3.16.1.12.5 LogFilter (Class)

This class is used to specify the criteria to be used when getting entries from the Communications Log. The caller would create an object of this type specifying the criteria that each log entry must match in order to be returned.

3.16.1.12.6 LogIterator (Class)

This class represents an iterator to iterate through a collection of log entries. If a retrieval request results in more data than is reasonable to transmit all at once, one clump of entries is returned at first, together with a LogIterator from which additional data can be requested, repeatedly, until all entries are returned or the user cancels the operation.

3.16.1.13 TrafficEventManagement (Class Diagram)

This class diagram contains all classes relating to Traffic Events

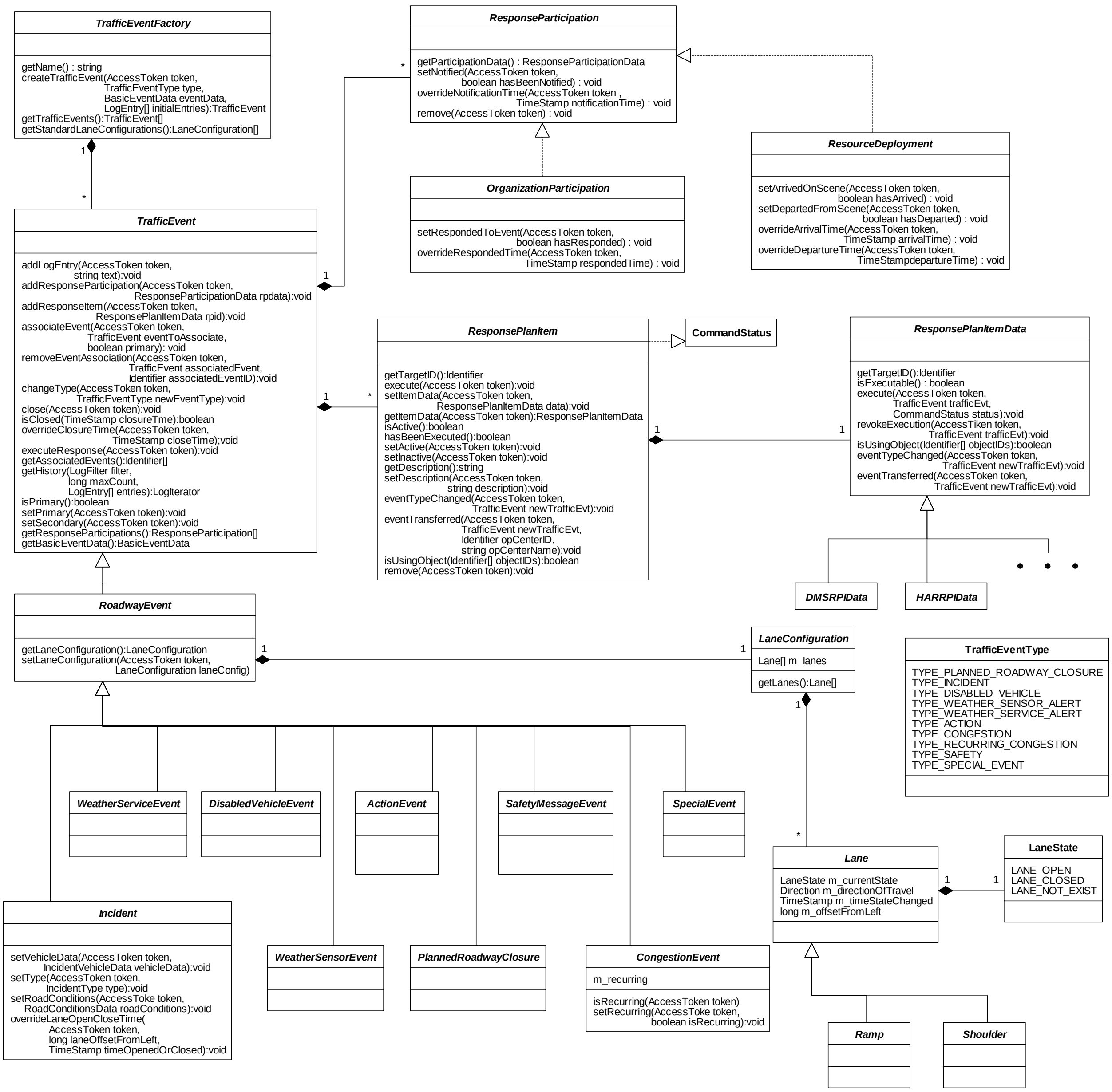

Figure 164. TrafficEventManagement (Class Diagram)

3.16.1.13.1 ActionEvent (Class)

This class models roadway events that require an operations center to take action but do not fit well into the other event categories. An example of this type of event would be debris in the roadway.

3.16.1.13.2 CommandStatus (Class)

The CommandStatus CORBA interface is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

3.16.1.13.3 CongestionEvent (Class)

This class models roadway congestion that may be tagged as recurring or non-recurring through the use of an attribute.

3.16.1.13.4 DisabledVehicleEvent (Class)

This class models disabled vehicles on the roadway.

3.16.1.13.5 DMSRPIData (Class)

The DMSRPIData class is an abstract class that describes a response plan item for a DMS. It contains the unique identifier of the DMS to contain the DMSMessage, and the DMSMessage itself.

3.16.1.13.6 HARRPIData (Class)

This class represents an item in a traffic event response plan that is capable of issuing a command to put a message on a HAR when executed. When the item is executed, it adds the message to the arbitration queue of the specified HAR. When the item is removed from the response plan (manually or implicitly through closing the traffic event) the item asks the HAR's arbitration queue to remove the message.

3.16.1.13.7 Incident (Class)

This class models objects representing roadway incidents. An incident typically involves one or more vehicles and roadway lane closures.

3.16.1.13.8 Lane (Class)

This class represents a single traffic lane at the scene of a RoadwayEvent.

3.16.1.13.9 LaneConfiguration (Class)

This class contains data that represents the configuration of the lanes.

3.16.1.13.10 LaneState (Class)

This enumeration lists the possible states that a traffic lane may be in.

3.16.1.13.11 OrganizationParticipation (Class)

This class is used to manage the data captured when an operator notifies another organization of a traffic event.

3.16.1.13.12 PlannedRoadwayClosure (Class)

This class models planned roadway closures such as road construction. This interface will be expanded in future releases to include interfacing with the EORS system.

3.16.1.13.13 Ramp (Class)

This class represents a ramp type traffic lane.

3.16.1.13.14 ResponseParticipation (Class)

This interface represents the involvement of one particular resource or organization in response to a particular traffic event.

3.16.1.13.15 ResponsePlanItem (Class)

Objects of this type can be executed as part of a traffic event response plan. A ResponsePlanItem can be executed by an operator, at which time it becomes the responsibility of the System to activate the item on the ResponseDevice as soon as it is appropriate.

3.16.1.13.16 ResponsePlanItemData (Class)

This class is a delegate used to perform the execute and remove tasks for the response plan item. Derived classes of this base class have specific implementations for the type of device the response plan item is used to control.

3.16.1.13.17 ResourceDeployment (Class)

This class is used to store the data captured when an operator deploys resources to the scene of a traffic event.

3.16.1.13.18 RoadwayEvent (Class)

This class models any type of incident that can occur on a roadway. This point in the heirarchy provides a break off point for traffic event types that pertain to other modals.

3.16.1.13.19 SafetyMessageEvent (Class)

This type of event is created by an operator when he/she would like to send a safety message to a device.

3.16.1.13.20 Shoulder (Class)

This class represents a shoulder type traffic lane.

3.16.1.13.21 SpecialEvent (Class)

This class models special events that affect roadway conditions such as a concert or professional sporting event.

3.16.1.13.22 TrafficEvent (Class)

Objects of this type represent traffic events that require action from system operators.

3.16.1.13.23 TrafficEventFactory (Class)

This interface is supported by objects that are capable of creating traffic event objects in the system.

3.16.1.13.24 TrafficEventType (Class)

This enum defines the types of traffic events that are supported by the system.

3.16.1.13.25 WeatherSensorEvent (Class)

This class models roadway weather events such as snow or fog that are reported by the system’s weather monitoring devices. Operators will need to manually enter the information in these events for this release. In future releases, these events will be automatically generated by the system.

3.16.1.13.26 WeatherServiceEvent (Class)

This class models roadway weather events such as snow or fog that are manually entered by an operator in response to receiving an alert from the national weather service.

3.16.1.14 TrafficEventManagement2 (Class Diagram)

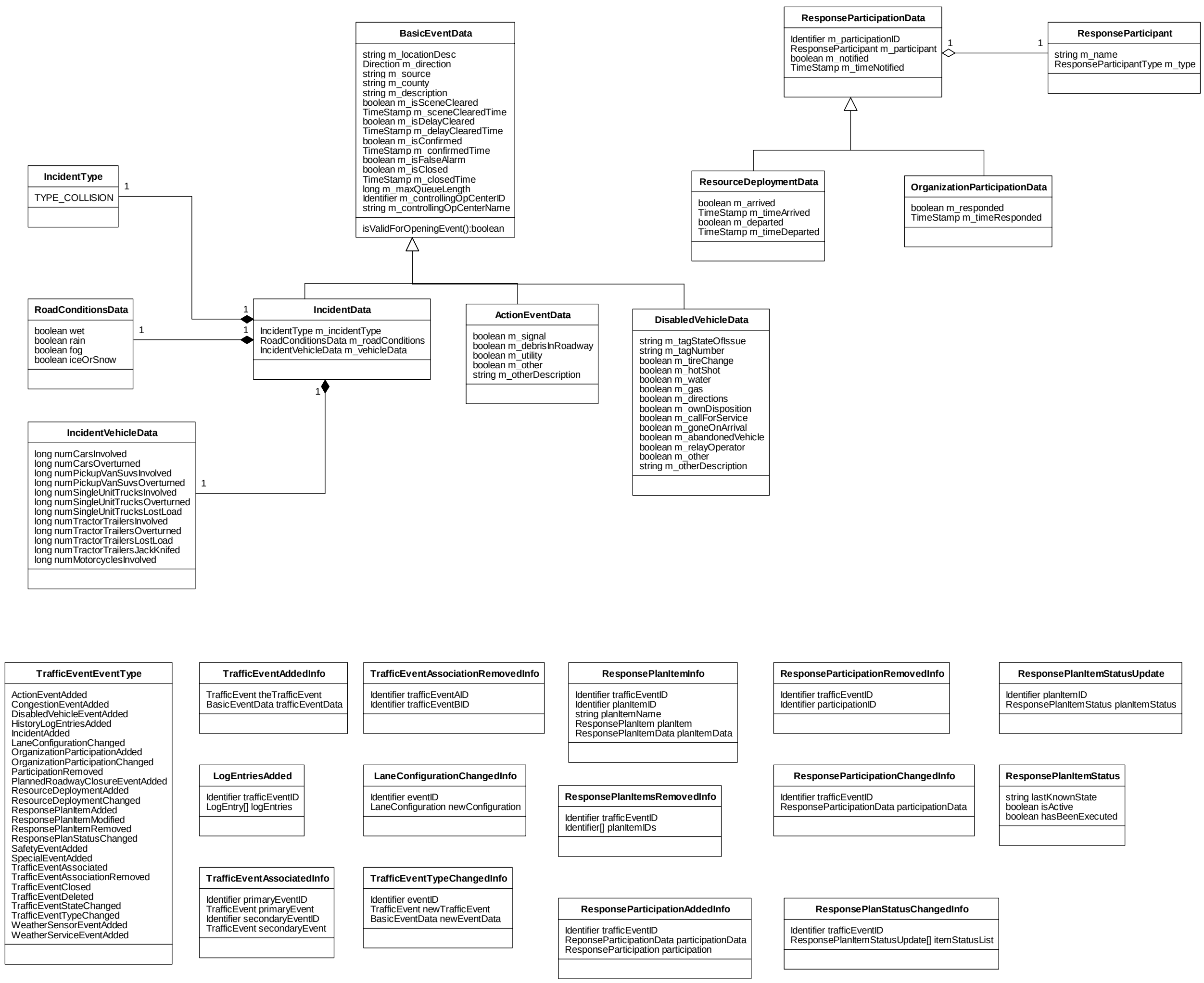

Figure 165. TrafficEventManagement2 (Class Diagram)

3.16.1.14.1 ActionEventData (Class)

This class represents all data specific to an Action event type traffic event.

3.16.1.14.2 BasicEventData (Class)

This class represents the data common to all traffic events. All derived data types will inherit all data shown in this class.

3.16.1.14.3 DisabledVehicleData (Class)

This class represents all data specific to a disabled vehicle traffic event.

3.16.1.14.4 IncidentData (Class)

This class represents data specific to an Incident type traffic event.

3.16.1.14.5 IncidentType (Class)

This enumeration lists all possible incident types.

3.16.1.14.6 IncidentVehicleData (Class)

This class represents the vehicles involved data for incidents. Its purpose is to simplify the exchange of data between GUI and server.

3.16.1.14.7 LaneConfigurationChangedInfo (Class)

This structure contains the data that is broadcast when the lane configuration of a traffic event is changed.

3.16.1.14.8 LogEntriesAdded (Class)

This structure contains the data that is broadcast when new entries are added to the event history log of a traffic event.

3.16.1.14.9 OrganizationParticipationData (Class)

This class represents the data required to describe an organization’s participation in the response to a traffic event.

3.16.1.14.10 ResourceDeploymentData (Class)

This class represents the data required to describe a resource’s participation in the response to a traffic event.

3.16.1.14.11 ResponseParticipant (Class)

The ResponseParticipant class is a non-behavioral structure that specifies a participant in a response.

3.16.1.14.12 ResponseParticipationData (Class)

This class contains all data pertinent to any class that represents a response participation.

3.16.1.14.13 ResponsePlanItemStatus (Class)

This stucture contains data that describes the current state of a response plan item.

3.16.1.14.14 ResponsePlanStatusChangedInfo (Class)

This structure contains the data that is broadcast when one or more response plan items in the response plan of a traffic event change state.

3.16.1.14.15 RoadConditionsData (Class)

This class represents the data necessary to describe the road conditions at the scene of a traffic event.

3.16.1.14.16 ResponseParticipationAddedInfo (Class)

This structure contains the data that is broadcast when a response participant is added to the response to a particular traffic event.

3.16.1.14.17 ResponseParticipationRemovedInfo (Class)

This structure contains the data that is broadcast when one or more response plan items are removed from a traffic event.

3.16.1.14.18 ResponseParticipationChangedInfo (Class)

This structure contains the data pushed in a CORBA event any time any type of response participation object changes state.

3.16.1.14.19 ResponsePlanItemInfo (Class)

This structure contains the data that is broadcast any time a new response plan item is added or an existing response plan item is modified.

3.16.1.14.20 ResponsePlanItemsRemovedInfo (Class)

This structure contains the data that is broadcast when one or more response plan items are removed from a traffic event.

3.16.1.14.21 ResponsePlanItemStatusUpdate (Class)

This structure contains data that describes a status change to a particular response plan item.

3.16.1.14.22 TrafficEventAddedInfo (Class)

This structure contains the data that is broadcast when a new traffic event is added to the system.

3.16.1.14.23 TrafficEventAssociatedInfo (Class)

This structure contains the data that is broadcast when two traffic events are associated.

3.16.1.14.24 TrafficEventAssociationRemovedInfo (Class)

This structure contains the data that is broadcast when the association between two traffic events is removed.

3.16.1.14.25 TrafficEventEventType (Class)

This enumeration defines the types of CORBA events that can be broadcast on a Traffic Event related CORBA Event channel.

3.16.1.14.26 TrafficEventTypeChangedInfo (Class)

This structure contains the data that is broadcast when a traffic event changes types. The traffic event object that represented the traffic event previously is removed from the system and is replaced by the newTrafficEvent reference contained in this structure. If the consumer of this CORBA event has stored any references to the traffic event previously, those references should be replaced with this new reference.

3.16.1.15 UserManagement (Class Diagram)

This class diagram contains the interfaces necessary to manage and utilize user profiles.

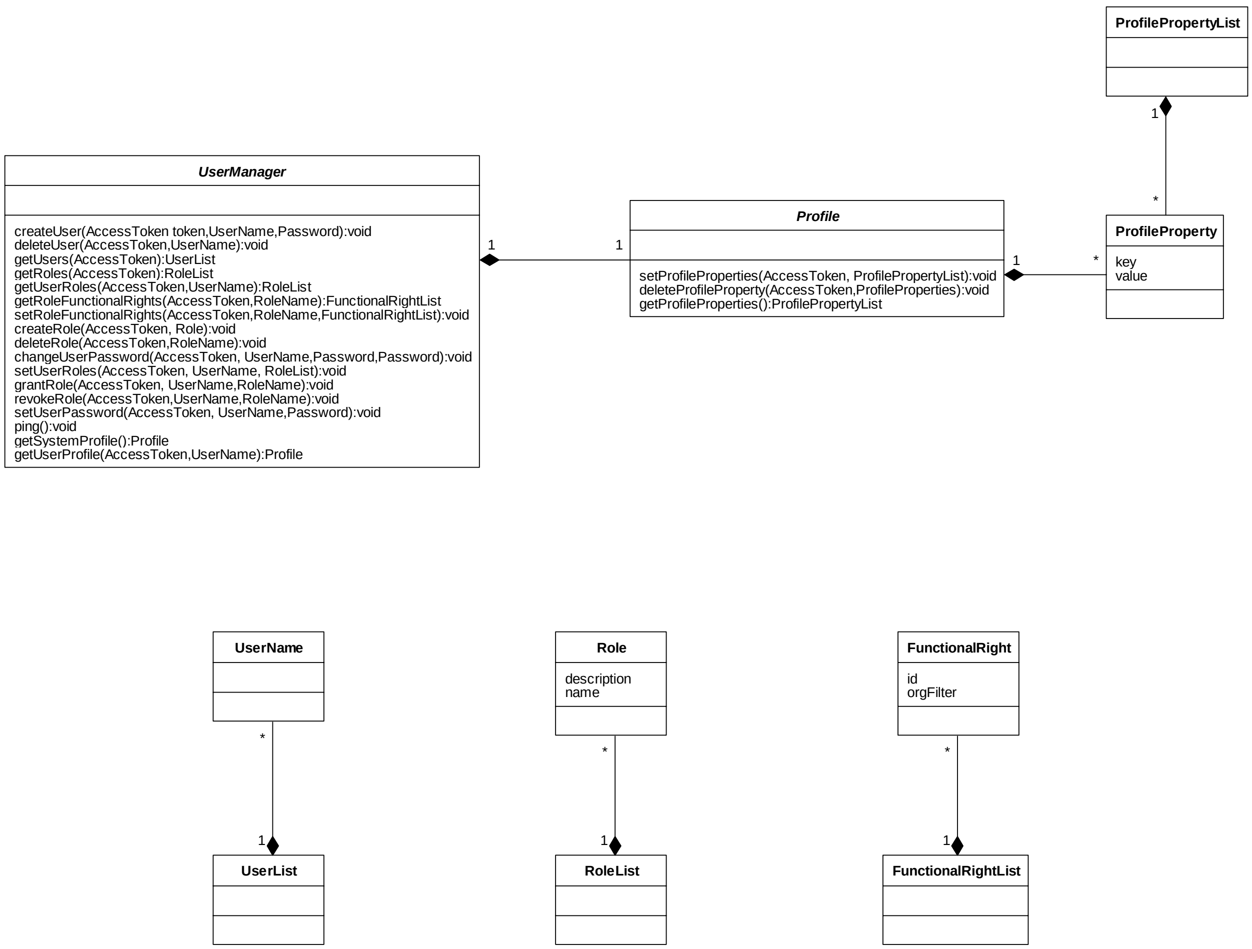

Figure 166. UserManagement (Class Diagram)

3.16.1.15.1 FunctionalRight (Class)

A functional right epresents a particular user capability. A functional right grants a particular capability to perform system functions. Each functional right may be limited by attaching the identifier of a particular organization to which this right is constrained. This capability allows an administrator to grant a particular Role the ability to modify only shared resources owned by the identified organization. The orgFilter identifier CHART2 will allow access to any organizations shared resources.

3.16.1.15.2 FunctionalRightList (Class)

A list of functional rights.

3.16.1.15.3 Profile (Class)

This class contains a set of user or administrator defined properties that are used to configure how the CHART II system behaves or presents information to a user.

3.16.1.15.4 ProfilePropertyList (Class)

A list of profile properties.

3.16.1.15.5 ProfileProperty (Class)

This class represents a key value pair that can be used to store system properties in the system database.

3.16.1.15.6 Role (Class)

A Role is a collection of functional rights. A Role can be granted to a user, thus granting the user all functional rights contained within the role.

3.16.1.15.7 RoleList (Class)

This structure contains a list of roles.

3.16.1.15.8 UserList (Class)

A list of user names.

3.16.1.15.9 UserName (Class)

This typedef defines the type of UserName fields used in system interfaces.

3.16.1.15.10 UserManager (Class)

The UserManager provides access to data dealing with user management. This includes users, roles, and functional rights. The UserManager is largely an interface to the User Management database tables.

3.17 TrafficEventModule

3.17.1 Classes

3.17.1.1 TrafficEventHierarchy (Class Diagram)

This diagram depicts the relationships between Traffic event related interfaces and their implementing classes. It does not show all possible traffic event types. Instead it shows a few of the many possible types for illustrative purposes. The main point of the diagram is to show that each TrafficEvent implementation object implements the corresponding CORBA interface and derives from the implementation object that implements its corresponding interface's parent interface.

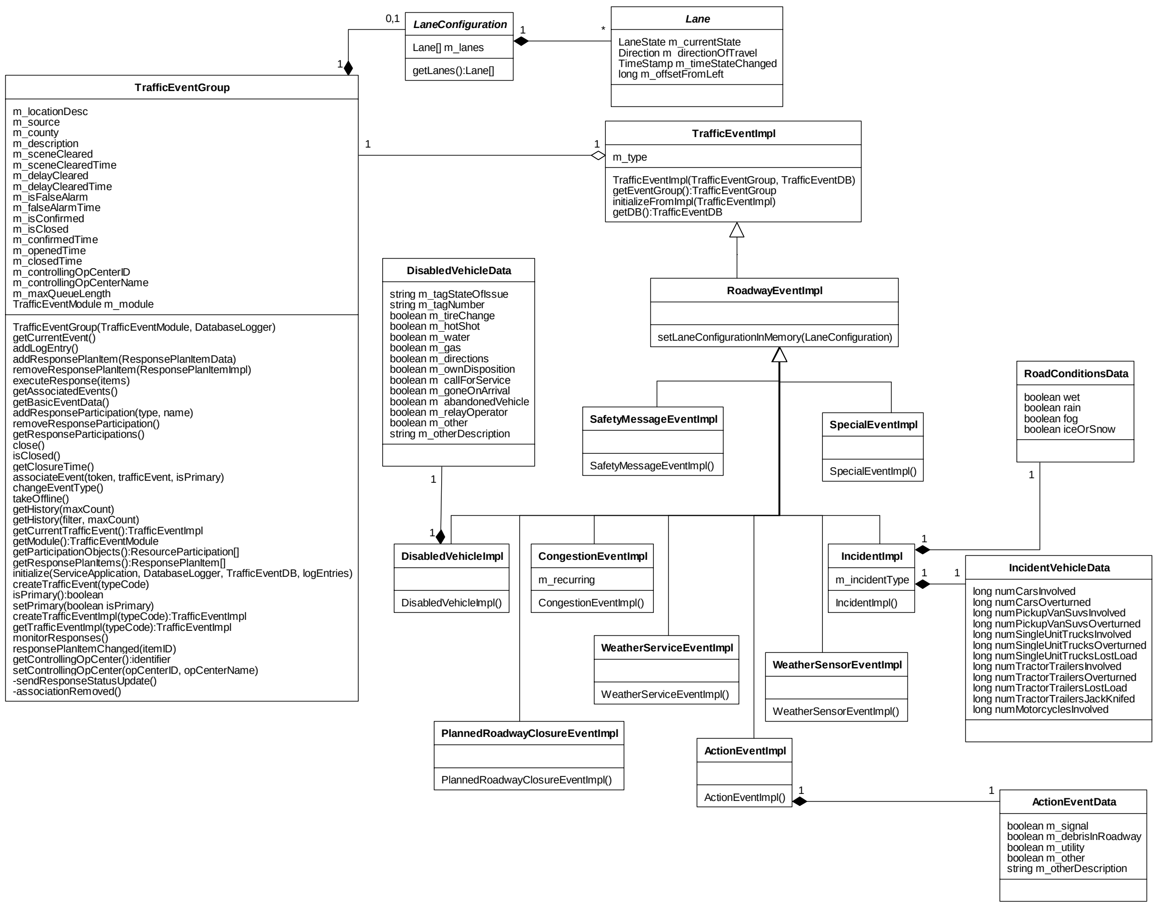

Figure 167. TrafficEventHierarchy (Class Diagram)

3.17.1.1.1 ActionEventData (Class)

This class represents all data specific to an Action event type traffic event.

3.17.1.1.2 ActionEventImpl (Class)

This class provides an implementation of the ActionEvent interface. Each ActionEventImpl contains a reference to a ActionEventData describing the event.

3.17.1.1.3 CongestionEventImpl (Class)

This class provides an implementation of the CongestionEvent interface. This contains the state variable to indicate if the event is a recurring event.

3.17.1.1.4 DisabledVehicleData (Class)

This class represents all data specific to a disabled vehicle traffic event.

3.17.1.1.5 DisabledVehicleImpl (Class)

This class provides an implementation of the DisabledVehicleEvent interface. Each DisableVehicleEventImpl contains a reference to DisabledVehicleData that describes the disabled vehicle details at the scene.

3.17.1.1.6 IncidentImpl (Class)

This class provides an implementation of the Incident interface. It contains state variables and processing that are unique to incident type traffic events.

3.17.1.1.7 IncidentVehicleData (Class)

This class represents the vehicles involved data for incidents. Its purpose is to simplify the exchange of data between GUI and server.

3.17.1.1.8 Lane (Class)

This class represents a single traffic lane at the scene of a RoadwayEvent.

3.17.1.1.9 LaneConfiguration (Class)

This class contains data that represents the configuration of the lanes.

3.17.1.1.10 PlannedRoadwayClosureEventImpl (Class)

This class provides an implementation of the PlannedRoadwayClosureEvent interface.

3.17.1.1.11 RoadConditionsData (Class)

This class represents the data necessary to describe the road conditions at the scene of a traffic event.

3.17.1.1.12 RoadwayEventImpl (Class)

This class provides an implementation of the RoadwayEvent interface. Each RoadwayEventImpl contains a reference to a LaneConfiguration that describes the lanes at the scene of the event.

3.17.1.1.13 SafetyMessageEventImpl (Class)

This class provides an implementation of the SafetyMessageEvent interface.

3.17.1.1.14 SpecialEventImpl (Class)

This class provides an implementation of the SpecialEvent interface.

3.17.1.1.15 TrafficEventImpl (Class)

This class provides an implementation of the TrafficEvent interface. It contains state variables and processing that common to all traffic events.

3.17.1.1.16 TrafficEventGroup (Class)

This class is used to group together different TrafficEvent objects that all represent the same traffic event that an operations center is working. A particular traffic event may initially be created as a particular type of event such as DisabledVehicleEvent and later be converted to another type of event such as Incident. The group stores all information that is common to all of these TrafficEvent objects that represent the same roadway event.

3.17.1.1.17 WeatherSensorEventImpl (Class)

This class provides an implementation of the WeatherSensorEvent interface.

3.17.1.1.18 WeatherServiceEventImpl (Class)

This class provides an implementation of the WeatherServiceEvent interface.

3.17.1.2 TrafficEventModuleClasses (Class Diagram)

This diagram shows traffic event related classes and interfaces.

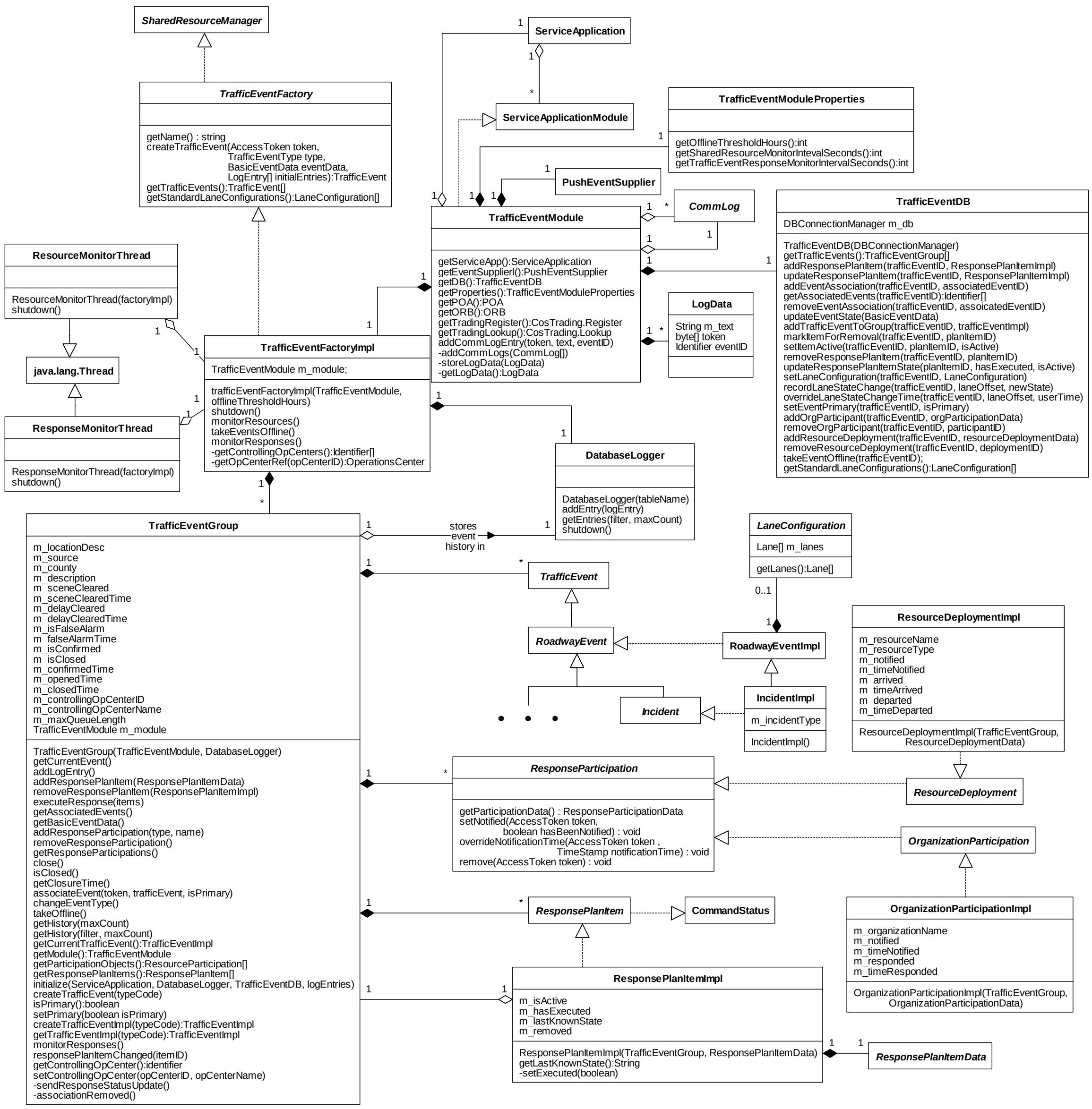

Figure 168. TrafficEventModuleClasses (Class Diagram)

3.17.1.2.1 CommLog (Class)

This class manages log entries. These can be general Communications Log entries or specific log entries for a specific Traffic Event. This class is the primary interface for the CommLog service. It is used to persist log entries in the CHART II system and retrieve them for review. Log entries can be created directly by users or indirectly as a result of manipulating Traffic Events.

3.17.1.2.2 CommandStatus (Class)

The CommandStatus CORBA interface is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

3.17.1.2.3 DatabaseLogger (Class)

This class represents a generic database logger that can be used to log and retrieve information from the database. This class also provides a mechanism for the user to filter and retrieve logs that meet specific criteria.

3.17.1.2.4 Incident (Class)

This class models objects representing roadway incidents. An incident typically involves one or more vehicles and roadway lane closures.

3.17.1.2.5 java.lang.Thread (Class)

This class represents a java thread of execution.

3.17.1.2.6 IncidentImpl (Class)

This class provides an implementation of the Incident interface. It contains state variables and processing that are unique to incident type traffic events.

3.17.1.2.7 LaneConfiguration (Class)

This class contains data that represents the configuration of the lanes.

3.17.1.2.8 LogData (Class)

3.17.1.2.9 OrganizationParticipation (Class)

This class is used to manage the data captured when an operator notifies another organization of a traffic event.

This class maintains a mapping between text messages and the corresponding audio clip file information. This is accomplished by maintaining a list of TreeMaps (one for each audio format supported) with text as key and audio clip information as the value. This class also helps manage the amount of hard drive space consumed by the audio clips by deleting the old clip files when the maximum cache size limit is reached. The maximum cache size limit can be set by the administrator using the system properties.

3.17.1.2.10 OrganizationParticipationImpl (Class)

This class provides an implementation of the OrganizationParticipation interface. Each instance represents a particular organization's participation activities in response to a particular traffic event.

3.17.1.2.11 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier's push rate.

3.17.1.2.12 ResponsePlanItem (Class)

Objects of this type can be executed as part of a traffic event response plan. A ResponsePlanItem can be executed by an operator, at which time it becomes the responsibility of the System to activate the item on the ResponseDevice as soon as it is appropriate.

3.17.1.2.13 ResponsePlanItemData (Class)

This class is a delegate used to perform the execute and remove tasks for the response plan item. Derived classes of this base class have specific implementations for the type of device the response plan item is used to control.

3.17.1.2.14 ResponsePlanItemImpl (Class)

This class provides an implementation of the ResponsePlanItem interface. Each instance represents one particular part of a response plan that can be in an executed, active or inactive state. This class also provides an implementation of the CommandStatus interface. This implies that devices that are activated on behalf of this traffic event can hold a copy of this object and call its update() method to provide a running status of the plan item as it changes.

3.17.1.2.15 ResourceDeployment (Class)

This class is used to store the data captured when an operator deploys resources to the scene of a traffic event.

3.17.1.2.16 ResourceDeploymentImpl (Class)

This class provides an implementation of the ResourceDeployment interface. Each instance represents a resource that has been deployed to the scene of a traffic event. This class contains the state data that describes the resource’s involvement in the traffic event.

3.17.1.2.17 ResourceMonitorThread (Class)

This thread will periodically call the traffic event factory implementation object and force it to monitor its shared resources.

3.17.1.2.18 ResponseMonitorThread (Class)

This thread will periodically call the traffic event factory implementation object and force it to notify each traffic event to monitor its response plan items for status changes.

3.17.1.2.19 ResponseParticipation (Class)

This interface represents the involvement of one particular resource or organization in response to a particular traffic event.

3.17.1.2.20 RoadwayEvent (Class)

This class models any type of incident that can occur on a roadway. This point in the heirarchy provides a break off point for traffic event types that pertain to other modals.

3.17.1.2.21 RoadwayEventImpl (Class)

This class provides an implementation of the RoadwayEvent interface. Each RoadwayEventImpl contains a reference to a LaneConfiguration that describes the lanes at the scene of the event.

3.17.1.2.22 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.17.1.2.23 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.17.1.2.24 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center. The shared resource manager is also responsible for periodically monitoring its shared resources to detect if the operations center controlling a resource doesn’t have at least one user logged into the system. When this condition is detected, the shared resource manager must push an event on the ResourceManagement event channel to notify others of this condition.

3.17.1.2.25 TrafficEvent (Class)

Objects of this type represent traffic events that require action from system operators.

3.17.1.2.26 TrafficEventDB (Class)

This class provides an interface for the traffic event module to utilize the database. The interface provides methods needed to store and retrieve TrafficEvent related information.

3.17.1.2.27 TrafficEventFactory (Class)

This interface is supported by objects that are capable of creating traffic event objects in the system.

3.17.1.2.28 TrafficEventFactoryImpl (Class)

This class is capable of creating a new TrafficEvent object in the system. Additionally, it acts as a manager of existing traffic event objects by performing calls on all traffic event objects such as shared resource or response plan monitoring.

3.17.1.2.29 TrafficEventGroup (Class)

This class is used to group together different TrafficEvent objects that all represent the same traffic event that an operations center is working. A particular traffic event may initially be created as a particular type of event such as DisabledVehicleEvent and later be converted to another type of event such as Incident. The group stores all information that is common to all of these TrafficEvent objects that represent the same roadway event.

3.17.1.2.30 TrafficEventModule (Class)

This class provides the resources and support functionality necessary to serve traffic event related objects in a service application. It implements the ServiceApplicationModule interface that allows it to be served from any ServiceApplication.

3.17.1.2.31 TrafficEventModuleProperties (Class)

This class provides operations for getting values in the service's java properties file.

3.17.2 Sequence Diagrams

3.17.2.1 TrafficEventModule:AddCommLogEntry (Sequence Diagram)

When a traffic event is opened, closed, or it changes types, it needs to add an entry to the communications log. This diagram depicts the fault tolerance built into this operation. When the TrafficEventModule is called to add an entry to the communications log, it will check if it has any cached entries that need to be added. These cached entries would be the result of prior calls that were not successful. If there are cached entries, the module will attempt to add them to the last communications log that was successfully used. If this is the first attempted use of a communications log or the attempt to use the last communications log fails, the module will search the trading service for all known communications logs. Each of these logs will be stored for future use. The module will then begin attempting to log all cached log data to each of the discovered communications logs until there are no more communications logs to try, or there are no more entries to log. If all communications logs are tried and the entry still could not be logged, the entry will be added to the cache and this process will repeat again the next time a comm log entry is attempted.

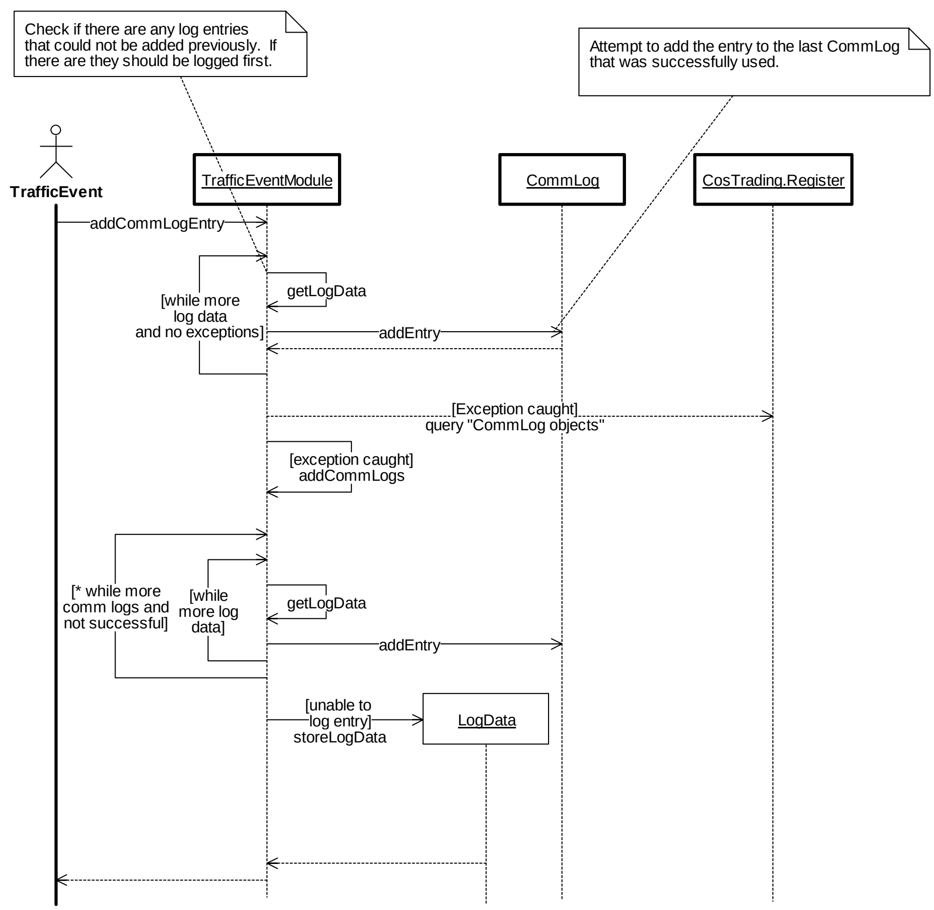

Figure 169. TrafficEventModule:AddCommLogEntry (Sequence Diagram)

3.17.2.2 TrafficEventModule:AddLogEntry (Sequence Diagram)

This diagram shows how an entry is added to a traffic event's history log. The TrafficEventImpl is called to add the log entry, and after checking the user's rights, it calls the TrafficEventGroup to add the entry. The TrafficEventGroup creates a new LogEntry and calls the DatabaseLogger to add the entry to the database. A CORBA event is then pushed through the event service, to update all of the GUIs with the new entry.

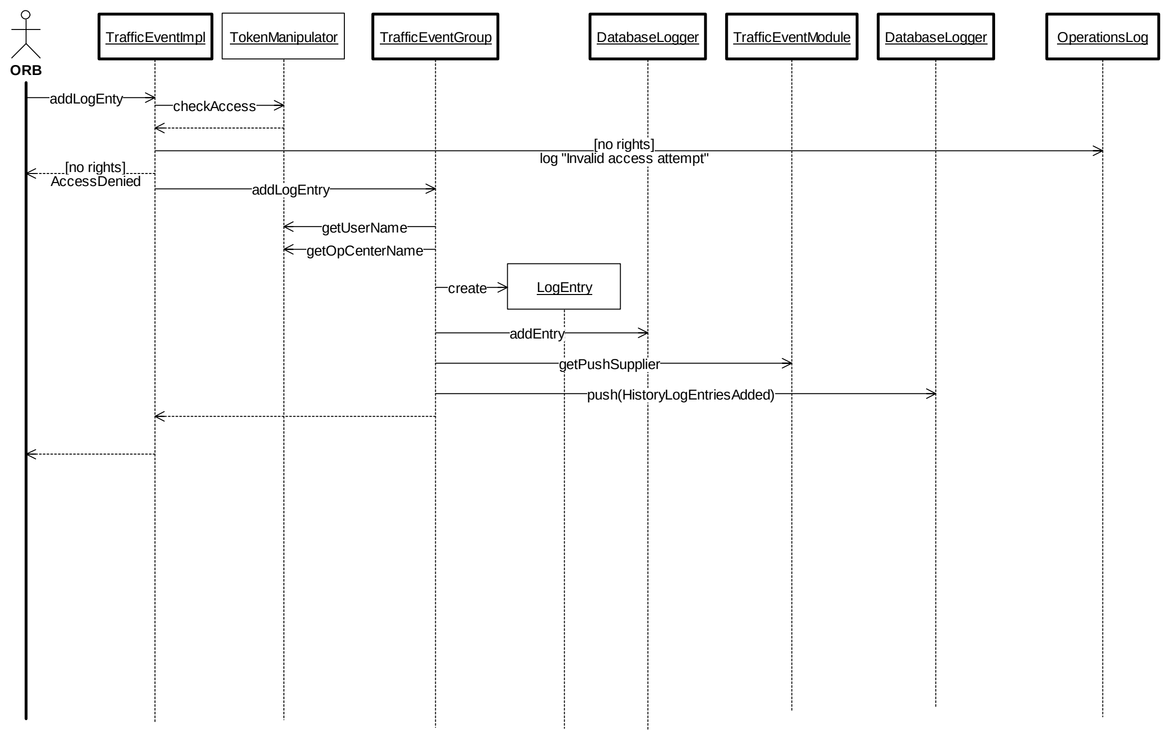

Figure 170. TrafficEventModule:AddLogEntry (Sequence Diagram)

3.17.2.3 TrafficEventModule:AddResponseItem (Sequence Diagram)

This diagram shows how a response item is added to a traffic event's response plan. The items can either be executable or non-executable (i.e., a placeholder containing only a target). The TrafficEvenImpl is called to add the ResponsePlanItem. After checking the user's rights, it calls the TrafficEventGroup to add the item. The TrafficEventGroup checks for existing ResponsePlanItems with the same target as the item being added. If an existing item is found and the new item is not executable, the new item is ignored. If an existing item is found and the new item is executable, the group sets the data in the existing ResponsePlanItem, which will overwrite the old data and cause the item's state to be "not executed" if it is already executed (see the sequence diagram SetMessageForUseInResponsePlan for details). Otherwise, if there was not already an existing item, a new ResponsePlanItemImpl is created, added to the database, and activated. A CORBA event is pushed to the event service to inform the GUIs of the new item, and entries are added to the traffic event's history log and the operations log.

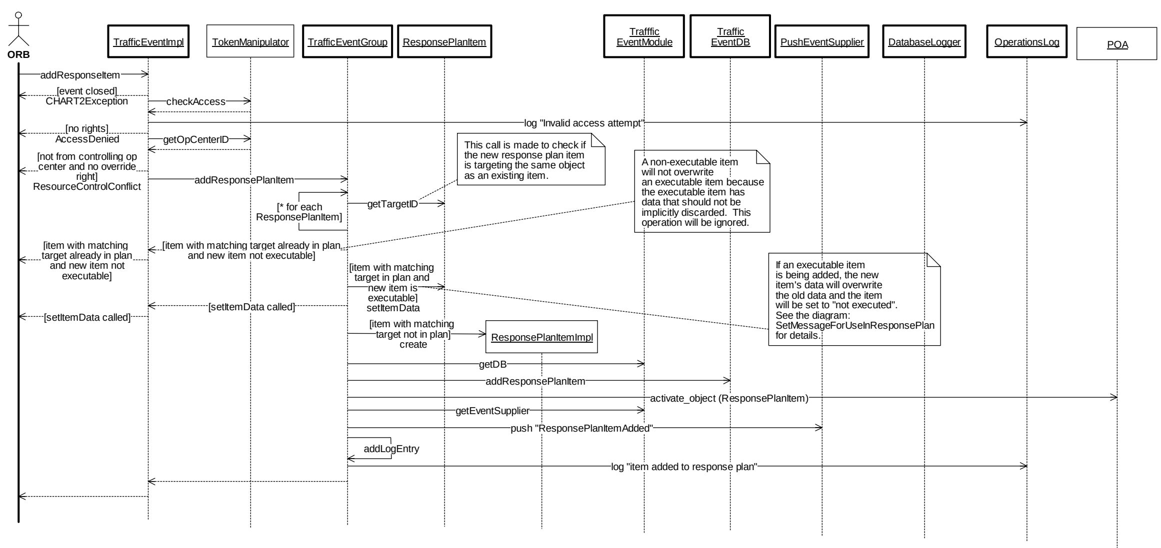

Figure 171. TrafficEventModule:AddResponseItem (Sequence Diagram)

3.17.2.4 TrafficEventModule:AddResponseParticipation (Sequence Diagram)

This diagram shows how a response participation is added to a traffic event. The TrafficEventImpl is called to add the response participation, and after checking the user's rights, calls the TrafficEventGroup to add the response participation. The TrafficEventGroup creates a new OrganizationParticipationImpl or a ResourceDeploymentImpl, then adds it to the database, activates the object to receive CORBA calls, and pushes a CORBA event through the event service so that all of the GUIs will be updated. An entry is also added to the traffic event's history log.

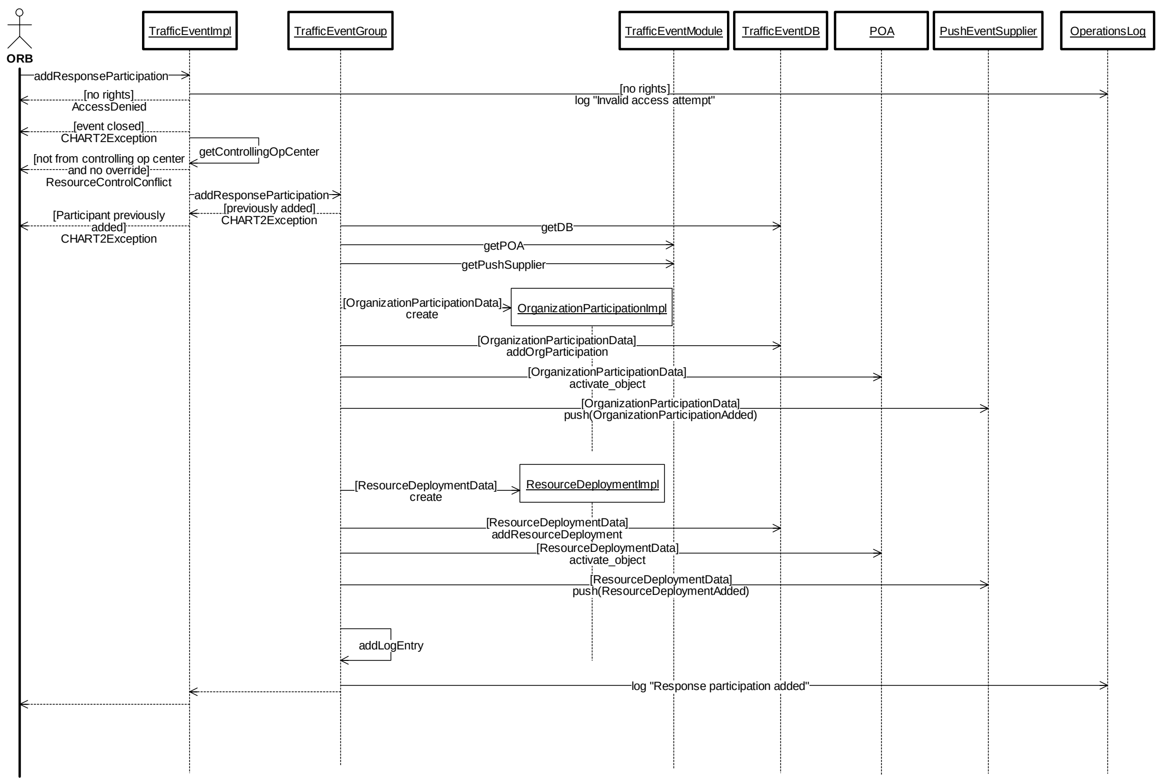

Figure 172. TrafficEventModule:AddResponseParticipation (Sequence Diagram)

3.17.2.5 TrafficEventModule:AssociateEvent (Sequence Diagram)

This diagram shows how a traffic event is associated to another traffic event. The TrafficEventImpl is called to associate the other event, and it calls the TrafficEventGroup after checking the rights. The TrafficEventGroup updates the database, adds entries to its history, and calls the other (secondary) event. The other event calls its event group, which marks itself as secondary, and updates the database. CORBA events are pushed by both TrafficEventGroups to notify the GUIs of the new association, and the new association is also stored in the database. Entries are added to the traffic events' histories and the operations log to record the change.

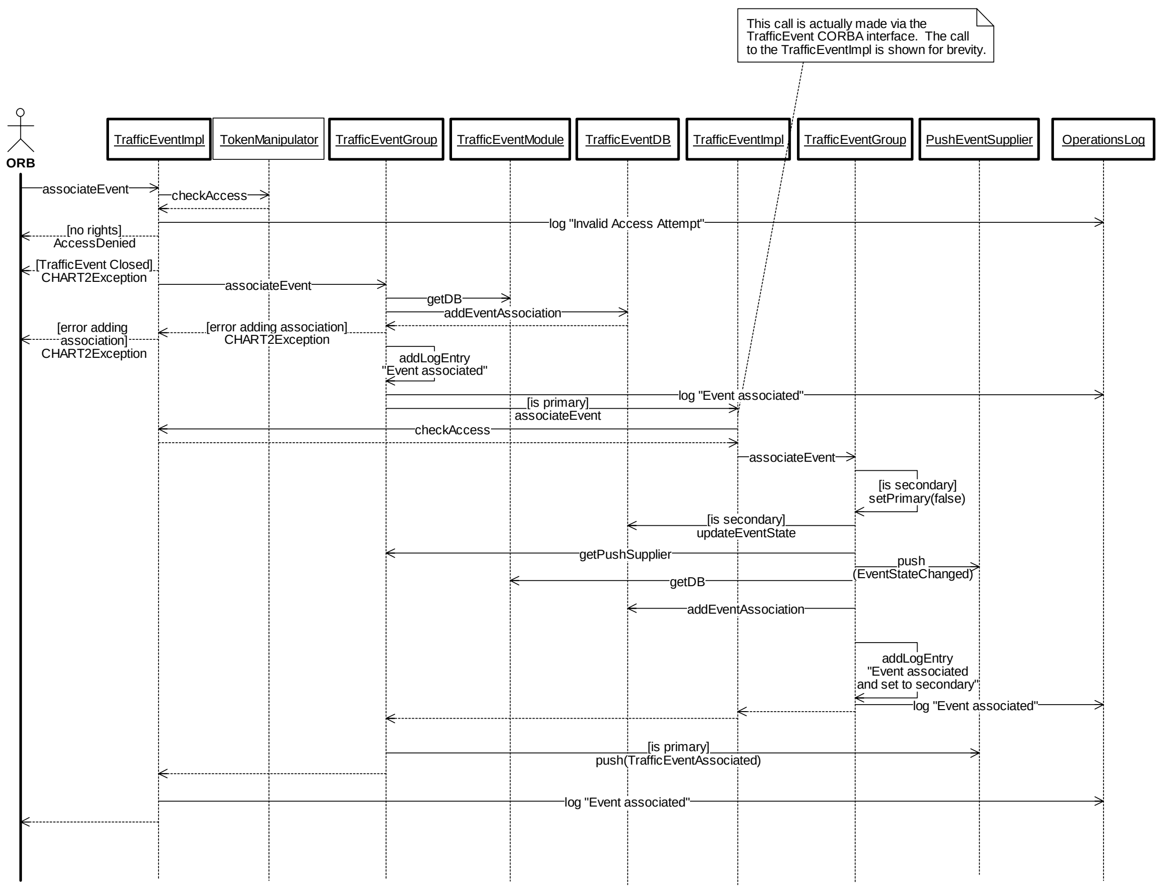

Figure 173. TrafficEventModule:AssociateEvent (Sequence Diagram)

3.17.2.6 TrafficEventModule:ChangeEventType (Sequence Diagram)

This diagram shows how the traffic event's type is changed. The TrafficEventImpl is called to change the type, and it calls the TrafficEventGroup after checking the user's rights. The TrafficEventGroup then searches in the event's previous history to find an event of the same type. If one is not found, a new TrafficEventImpl is created and initialized from the existing TrafficEventImpl, then it is added to the TrafficEventGroup and the database. Then it gets the old lane configuration from the TrafficEventGroup and sets it into the new TrafficEventImpl, if it's a RoadwayEvent. Then all of the ResponsePlanItems are notified of the new TrafficEvent so that they can switch their references to use the new event. The old TrafficEvent is withdrawn from the trading service and the new TrafficEvent is published in the trading service, and a CORBA event is pushed to update the GUIs. Entries are added to the traffic event's history log, the communications log, and the operations log, and the old TrafficEvent is deactivated.

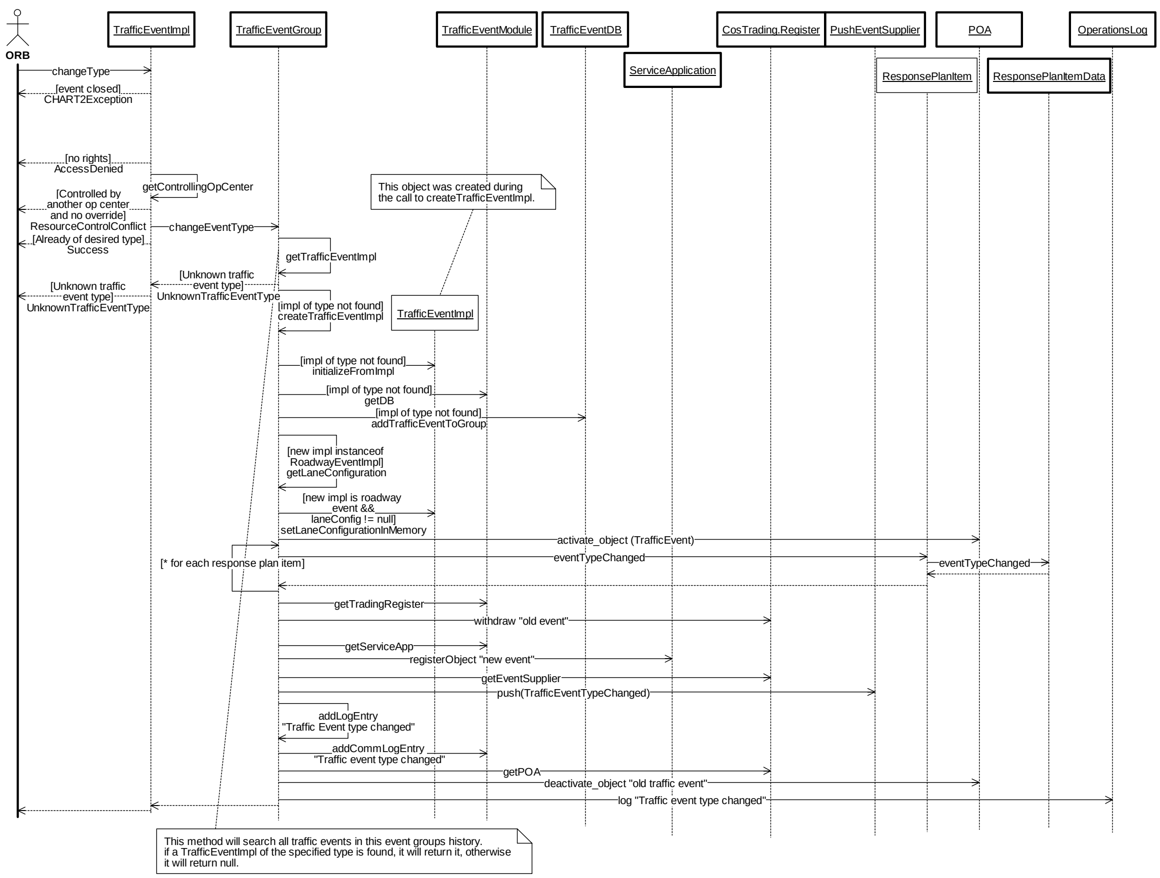

Figure 174. TrafficEventModule:ChangeEventType (Sequence Diagram)

3.17.2.7 TrafficEventModule:CloseEvent (Sequence Diagram)

This diagram shows what happens when a traffic event is closed. The TrafficEventImpl is called to close the event. After checking the user's rights, it calls the TrafficEventGroup to close the event. The group updates the event state in the database, and removes all of the ResponsePlanItems from the event. Entries are added to the traffic event's history, the communications log, and the operations log.

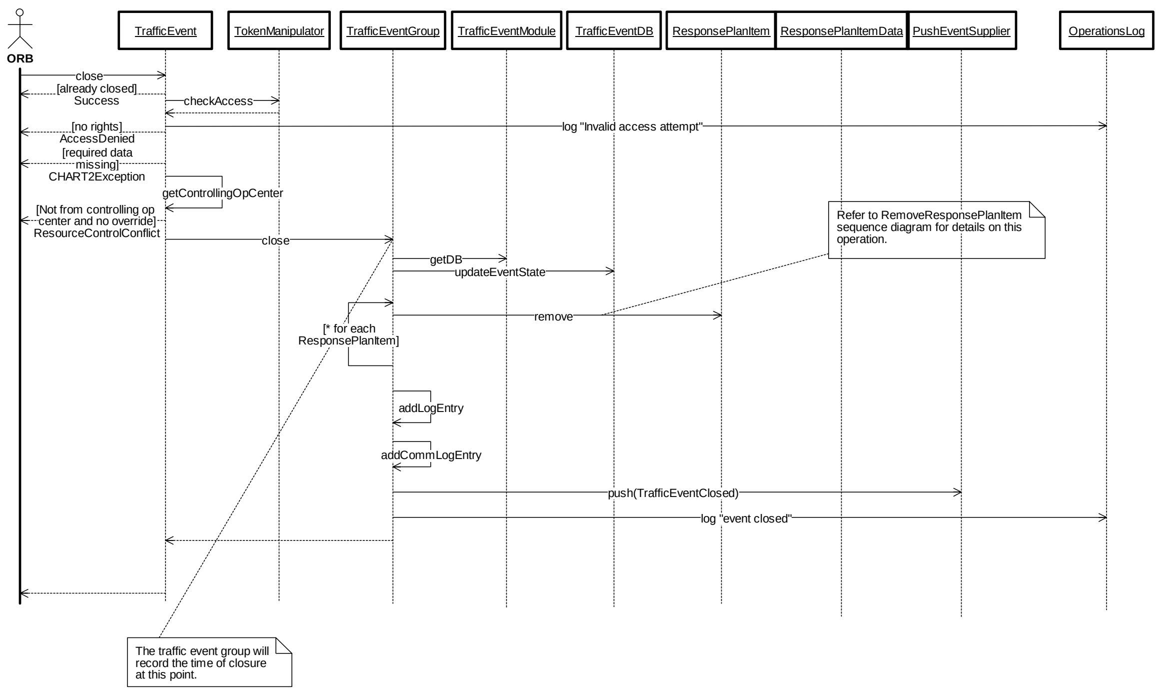

Figure 175. TrafficEventModule:CloseEvent (Sequence Diagram)

3.17.2.8 TrafficEventModule:CreateTrafficEvent (Sequence Diagram)

This diagram shows how a new traffic event is created. The TrafficEventFactoryImpl is called to create the new traffic event. After checking the user's rights, it creates a new TrafficEventGroup and calls it to create the appropriate type of TrafficEventImpl, based on the type of BasicTrafficEventData that is passed in. Then the factory calls the TrafficEventGroup to initialize. This adds any initial entries to the traffic event's history log, activates the TrafficEvent object, and publishes it in the trading service. It also adds entries to the communications log and the operations log, and pushes a CORBA event through the event service to inform the GUIs of the creation of the new event.

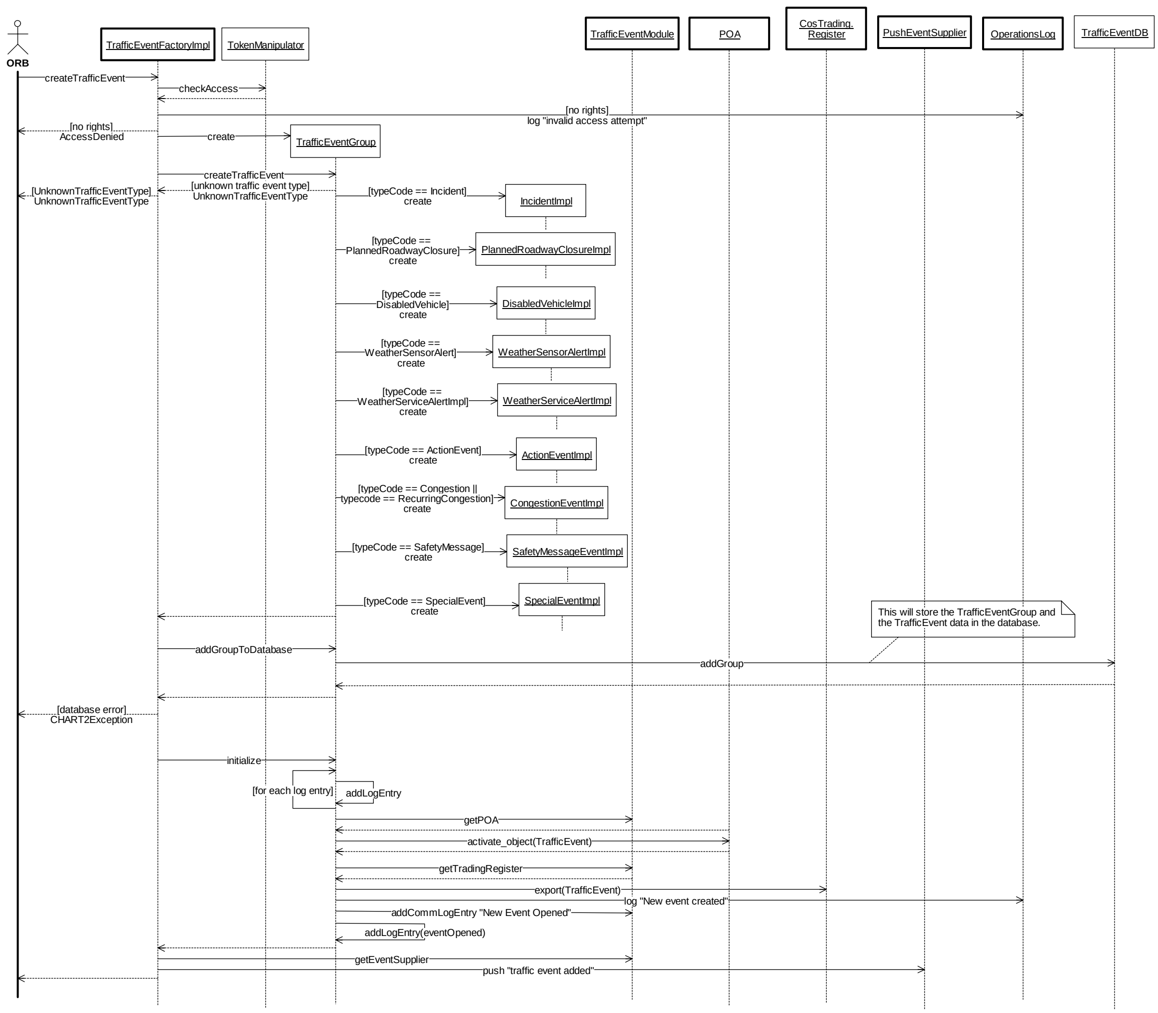

Figure 176. TrafficEventModule:CreateTrafficEvent (Sequence Diagram)

3.17.2.9 TrafficEventModule:ExecuteResponse (Sequence Diagram)

This diagram shows how a traffic event's response plan is executed. The TrafficEventImpl is called to execute the response. It checks the user's rights and then calls the TrafficEventGroup to execute the response. The TrafficEventGroup calls each ResponsePlanItem's execute method. See the ExecuteResponsePlanItem sequence diagram for details on how each response plan item is executed. The ResponseMonitorThread will be running in the background, and will periodically cause the factory to check all of the TrafficEventGroups for changes in the response plan item status. When prompted by this thread, each TrafficEventGroup will push a CORBA event to notify the GUIs if any of its response plan items have changed state.

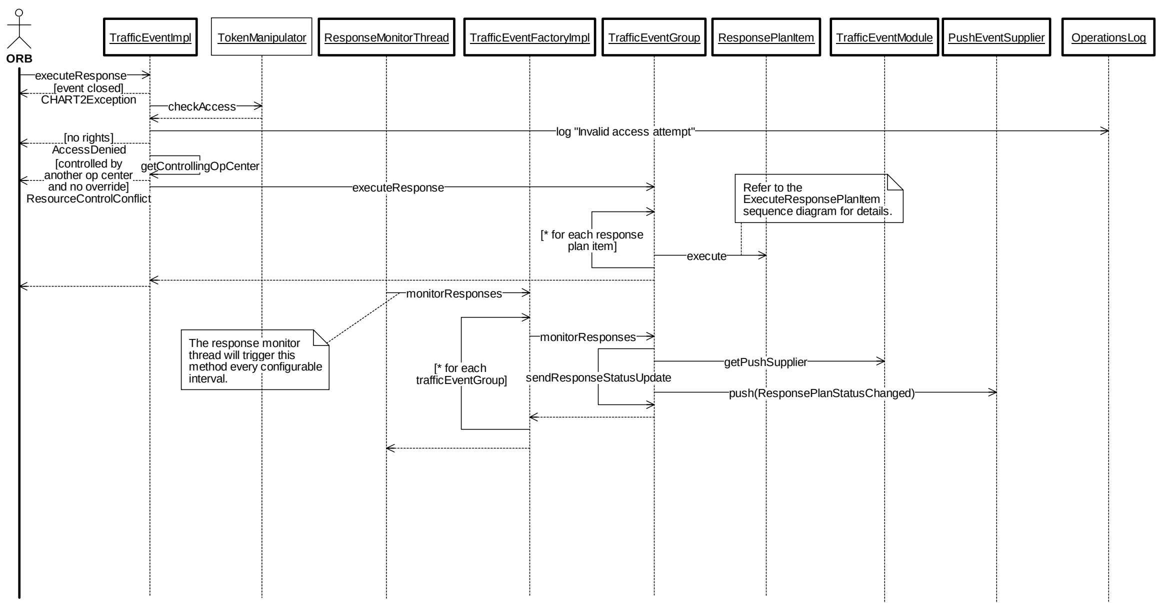

Figure 177. TrafficEventModule:ExecuteResponse (Sequence Diagram)

3.17.2.10 TrafficEventModule:ExecuteResponsePlanItem (Sequence Diagram)

This diagram shows what happens when a response plan item is executed, either individually or when a traffic event's response plan is executed. The user's rights are checked, and then the ResponsePlanItemImpl calls the ResponsePlanItemData to execute the item. The specific type of ResponsePlanItemData will call the appropriate target and the request to activate the message will be queued. Then the ResponsePlanItemImpl is marked as "executed", and the TrafficEventGroup is notified of the change in the item. The database is updated and an entry is added to the traffic event group's history log. The TrafficEventGroup will periodically be called on a background thread to push a CORBA event for any of its ResponsePlanItems that have changed state.

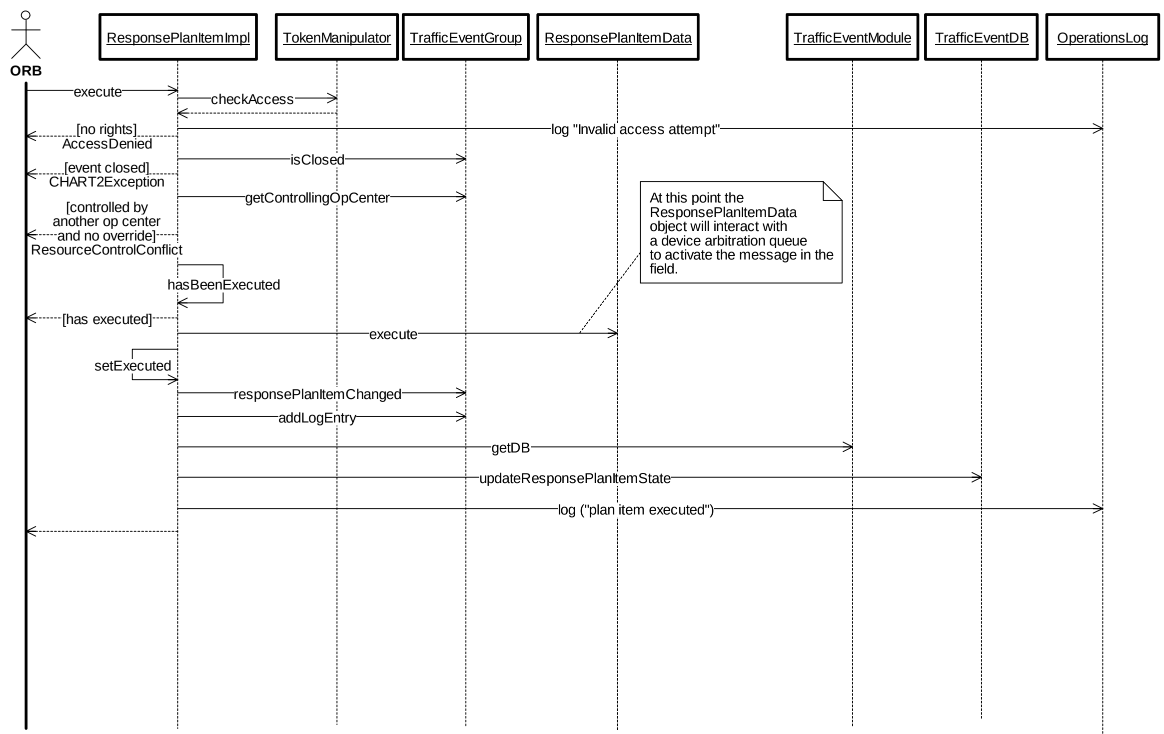

Figure 178. TrafficEventModule:ExecuteResponsePlanItem (Sequence Diagram)

3.17.2.11 TrafficEventModule:GetEventHistoryText (Sequence Diagram)

This diagram shows how entries are retrieved from the traffic event's history log. The TrafficEventImpl is called to get the event history. It checks the user's rights, then calls the TrafficEventGroup, which calls the DatabaseLogger to get the entries. See the sequence diagram DatabaseLogger:getEntries for more details.

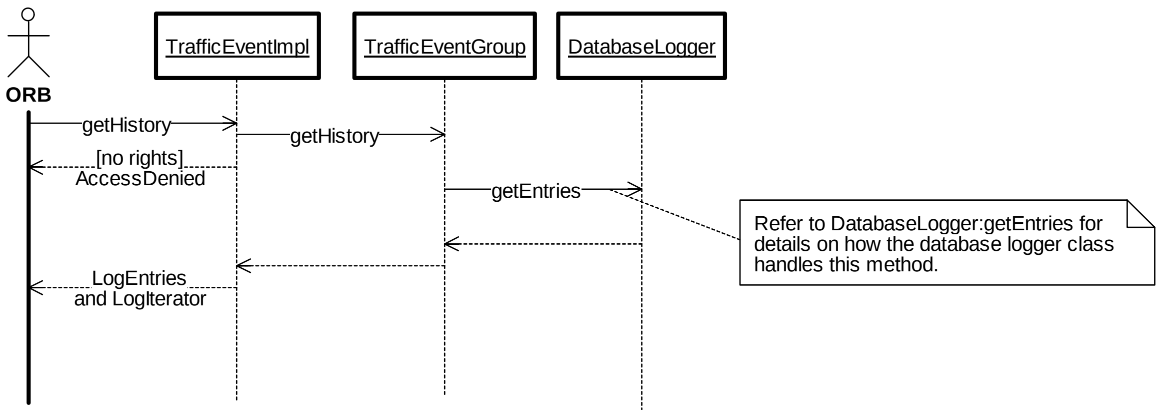

Figure 179. TrafficEventModule:GetEventHistoryText (Sequence Diagram)

3.17.2.12 TrafficEventModule:Initialize (Sequence Diagram)

This diagram shows what happens when the TrafficEventModule is initialized. The ServiceApplication calls the TrafficEventModule to initialize, which reads in the properties from a file, overriding the default properties. It creates an event channel for traffic events and publishes the channel in the trading service so that other applications can see it. It creates a TrafficEventDB object to handle all of the database calls, and a TrafficEventFactoryImpl object to manage the traffic events. The TrafficEventFactoryImpl creates a DatabaseLogger for logging the traffic event's history log, then calls the TrafficEventDB to load the TrafficEventGroup objects from the database. Then for each group it will activate the current TrafficEvent, the ResponseParticipation objects, and the ResponsePlanItem objects. The TrafficEvent is exported to the trading service. The resource monitor thread and the response monitor thread are created, and the TrafficEventFactory is exported to the trading service.

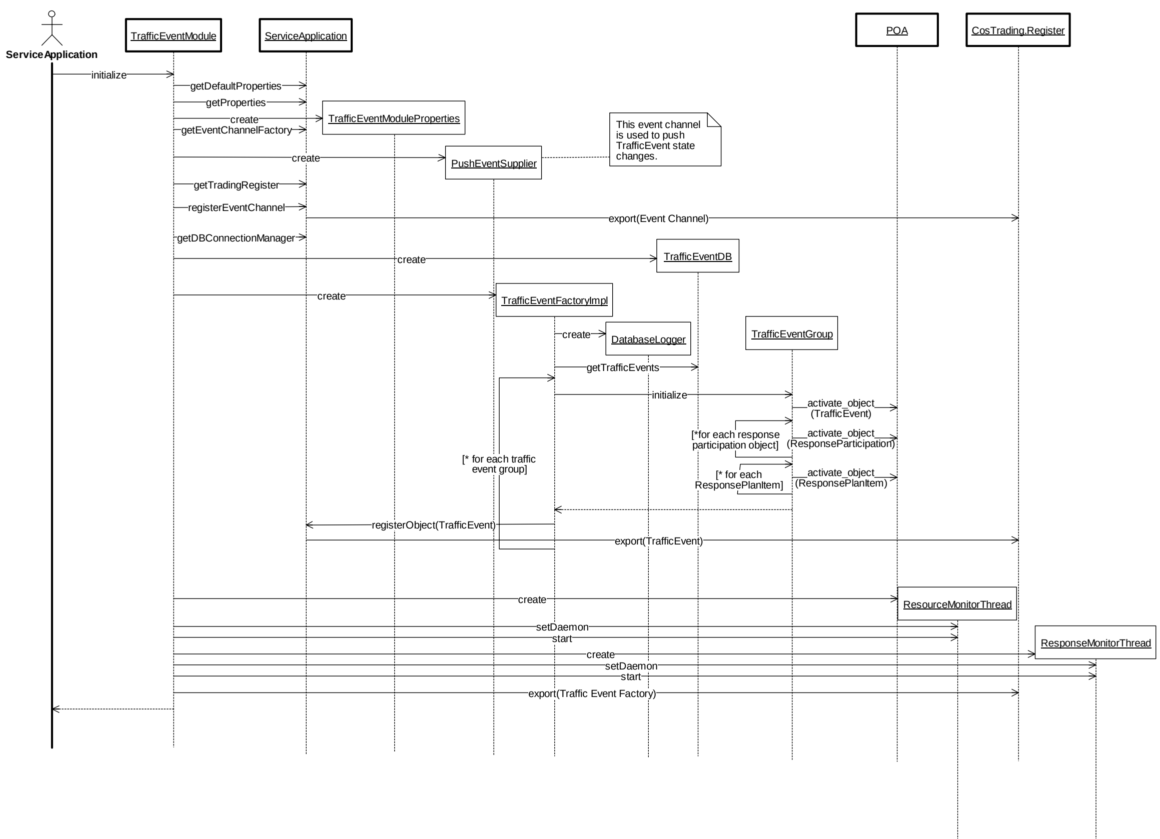

Figure 180. TrafficEventModule:Initialize (Sequence Diagram)

3.17.2.13 TrafficEventModule:MonitorControlledResources (Sequence Diagram)

This diagram shows the periodic maintenance of the traffic events—the monitoring of the controlling operations center, and the removal of the traffic events from the system. When the ResourceMonitorThread calls the factory to monitor the resources, the factory first gets all of the controlling operations centers for all traffic events. If it does not have references for all of the operations centers' IDs, it will query the OperationsCenter object from the trading service. Then it asks each OperationsCenter how many users are logged in. If no users are logged in, it pushes a CORBA event indicating that shared resources need to be transferred to another operations center. The ResourceMonitorThread will also call the factory to check if events need to be removed from the system. The factory asks each closed traffic event for its closure time and determines whether it has been closed long enough to remove it from the system. If a traffic event is removed, the database is updated, the offer is withdrawn from the tradiing service, the CORBA object is deactivated, and a CORBA event is pushed on the event channel indicating that the traffic event was just deleted.

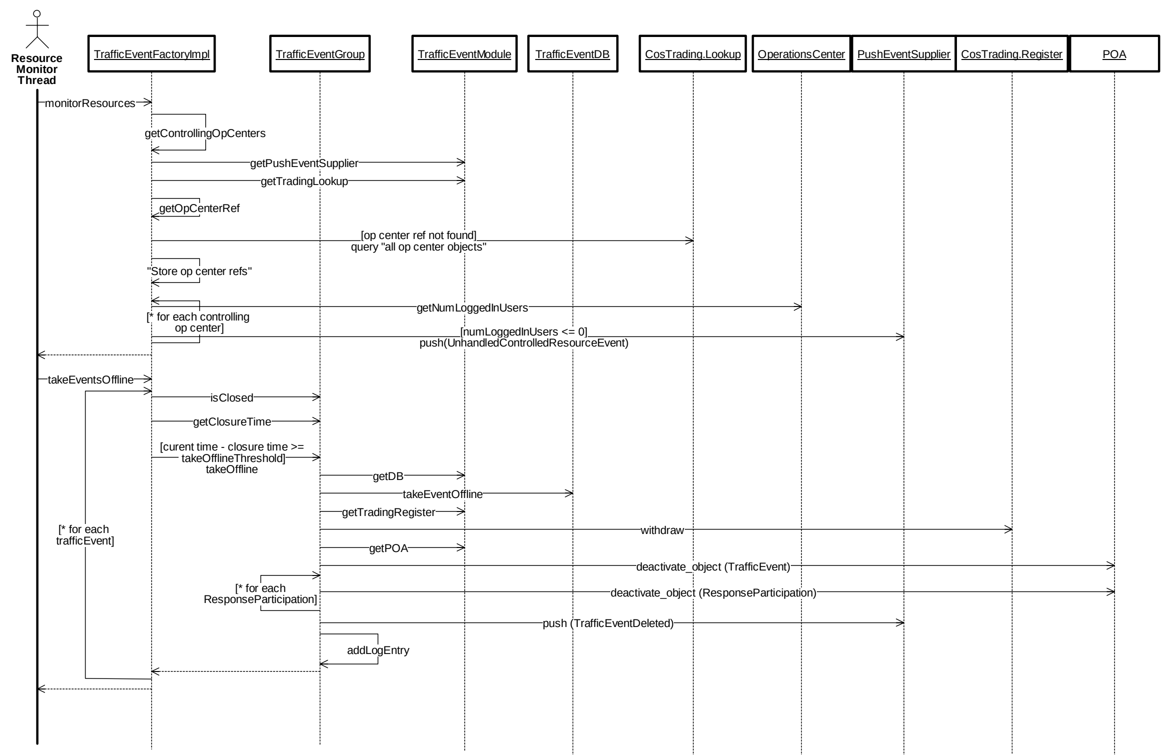

Figure 181. TrafficEventModule:MonitorControlledResources (Sequence Diagram)

3.17.2.14 TrafficEventModule:RemoveEventAssociation (Sequence Diagram)

This diagram shows what happens when a traffic event association is removed. One of the TrafficEventImpl objects is called to remove the association. It checks the user's rights and removes the association from its TrafficEventGroup and from the database and pushes an event. It also calls the associated event to remove the association from it. The associated event does the same thing, but when it calls back to the first TrafficEvent, the association has already been removed so it returns an exception to the second TrafficEvent and the association removal is complete.

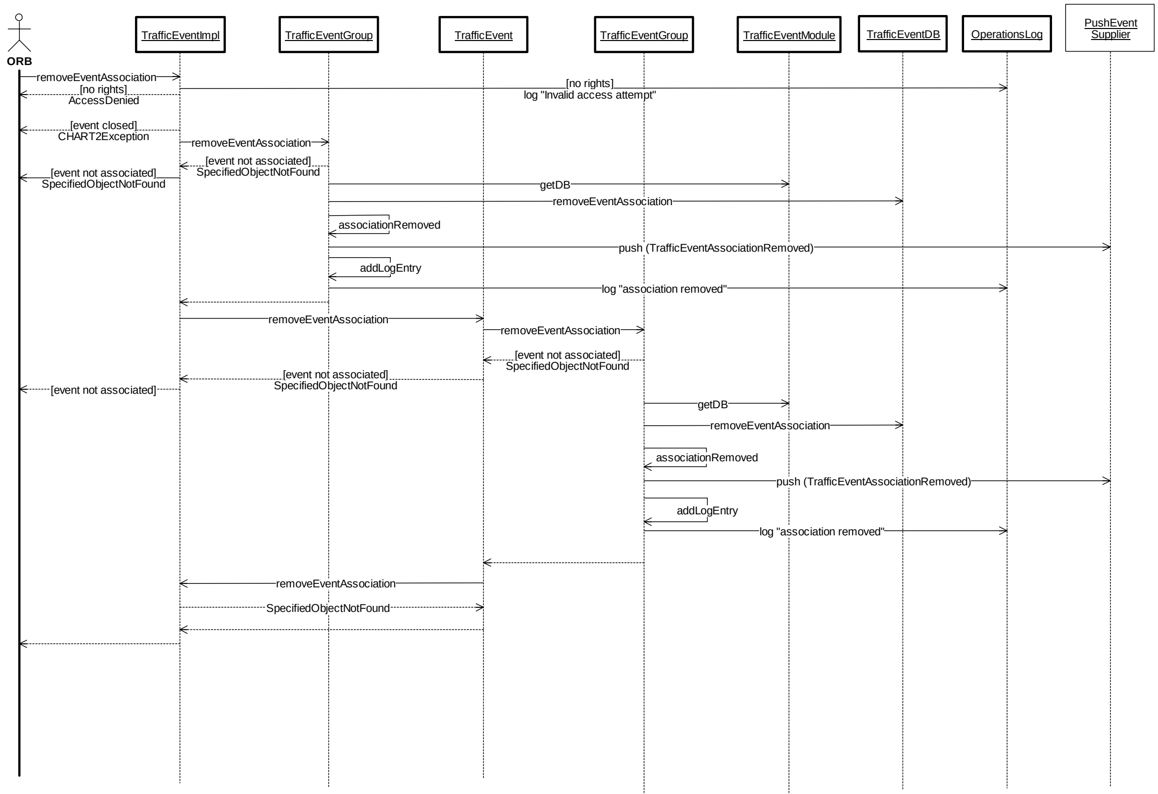

Figure 182. TrafficEventModule:RemoveEventAssociation (Sequence Diagram)

3.17.2.15 TrafficEventModule:RemoveResponseParticipation (Sequence Diagram)

This diagram shows how a response participation is removed from a traffic event. The ResponseParticipationImpl is called to remove itself. After checking the user's rights, it calls the TrafficEventGroup that is attached to and asks it to remove the participation. The TrafficEventGroup removes it from the database, deactivates the object, pushes a CORBA event to the event service, and adds entries to the event history log and operations log.

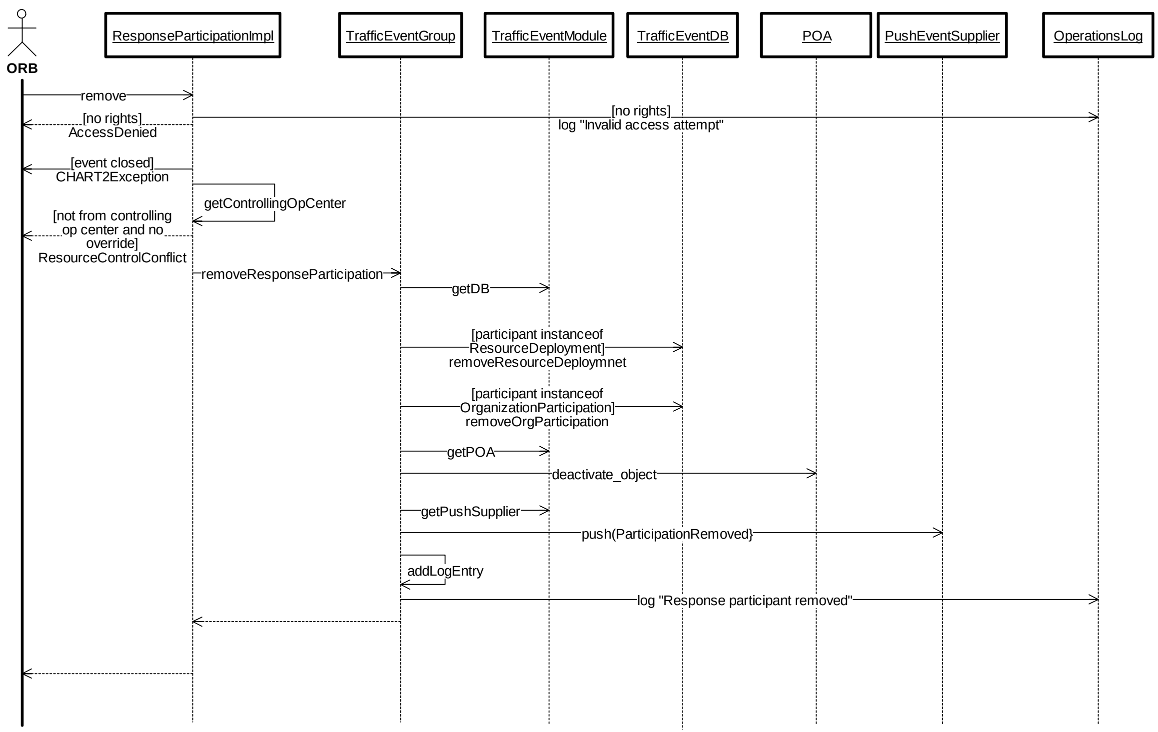

Figure 183. TrafficEventModule:RemoveResponseParticipation (Sequence Diagram)

3.17.2.16 TrafficEventModule:RemoveResponsePlanItem (Sequence Diagram)

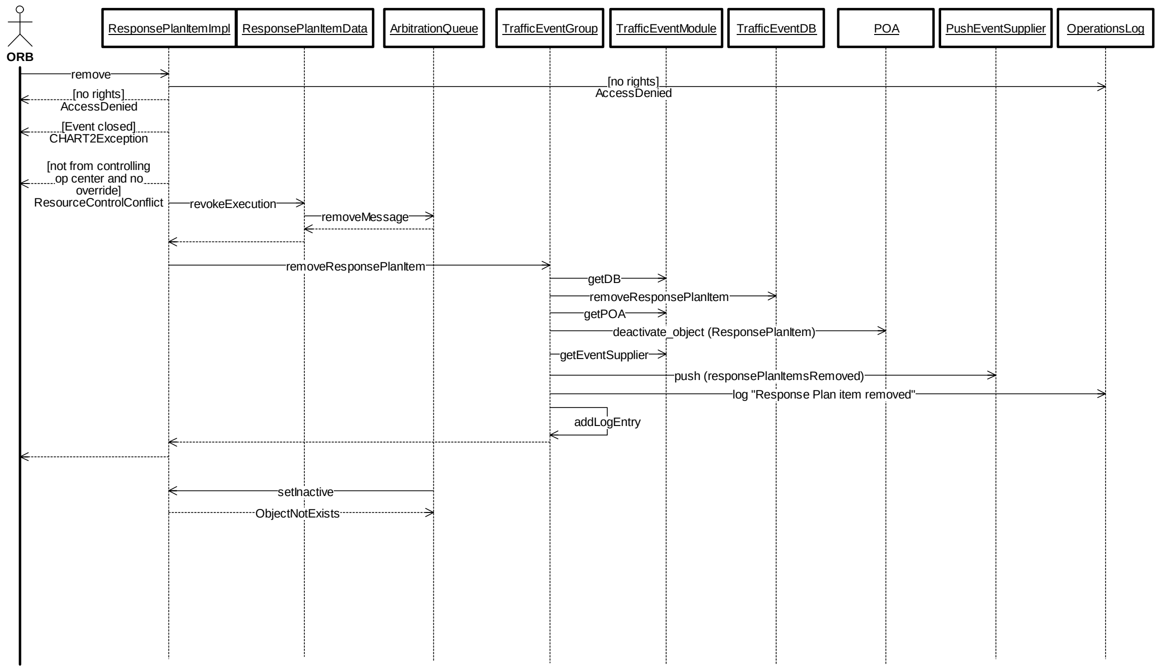

Figure 184. TrafficEventModule:RemoveResponsePlanItem (Sequence Diagram)

3.17.2.17 TrafficEventModule:SetLaneConfiguration (Sequence Diagram)

This diagram shows how the lane configuration is set for a roadway event. The RoadwayEventImpl is called to set the lane configuration. After checking the user's rights, it gets the old lane configuration and compares it to the new configuration. If there is a change in a lane's state, it records the state change in the database and a log entry is added to the traffic event's history log. Then a CORBA event is pushed indicating that the lane configuration has been set and entries are added to the traffic event's history log and operations log.

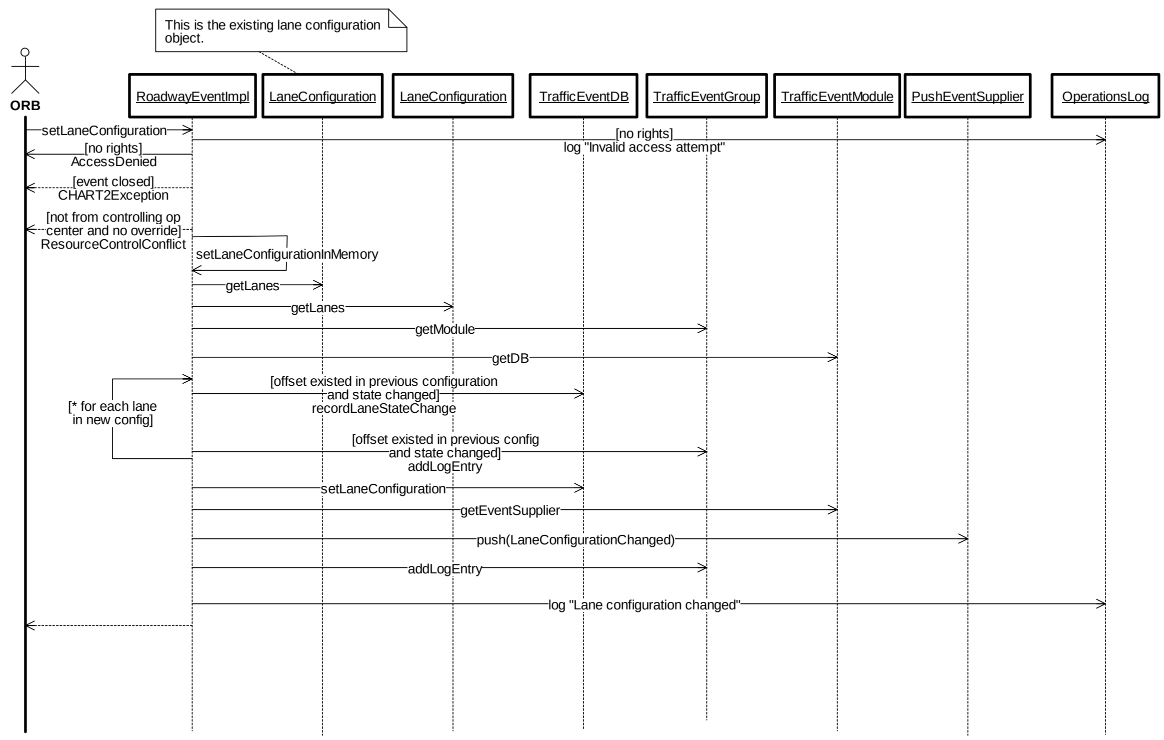

Figure 185. TrafficEventModule:SetLaneConfiguration (Sequence Diagram)

3.17.2.18 TrafficEventModule:SetMessageForUseInResponsePlan (Sequence Diagram)

This diagram shows how a message is modified within an existing response plan item. The ResponsePlanItemImpl is called to set the item data. After checking the user's rights, it marks the response plan item as being "not executed". It updates the plan item in the database and pushes a CORBA event via the event service indicating that the response plan item has changed. Entries are added to the traffic event's history log and the operations log.

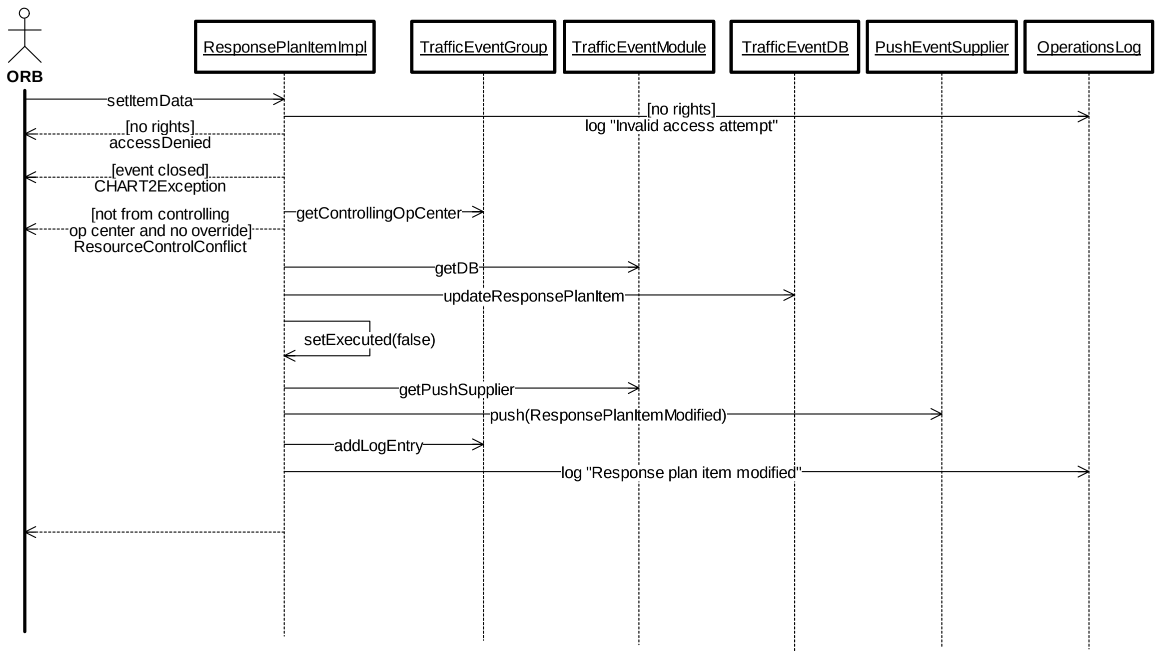

Figure 186. TrafficEventModule:SetMessageForUseInResponsePlan (Sequence Diagram)

3.17.2.19 TrafficEventModule:Shutdown (Sequence Diagram)

This diagram shows what happens at shutdown. The TrafficEventModule is called to shut down, and it calls the TrafficEventFactoryImpl, which calls all of the TrafficEventGroups. Each TrafficEventGroup deactivates the current TrafficEvent and all of its ResponseParticipation objects and ResponsePlanItem objects. Then the factory shuts down the resource monitor thread. The module deactivates the TrafficEventFactory object and shuts down.

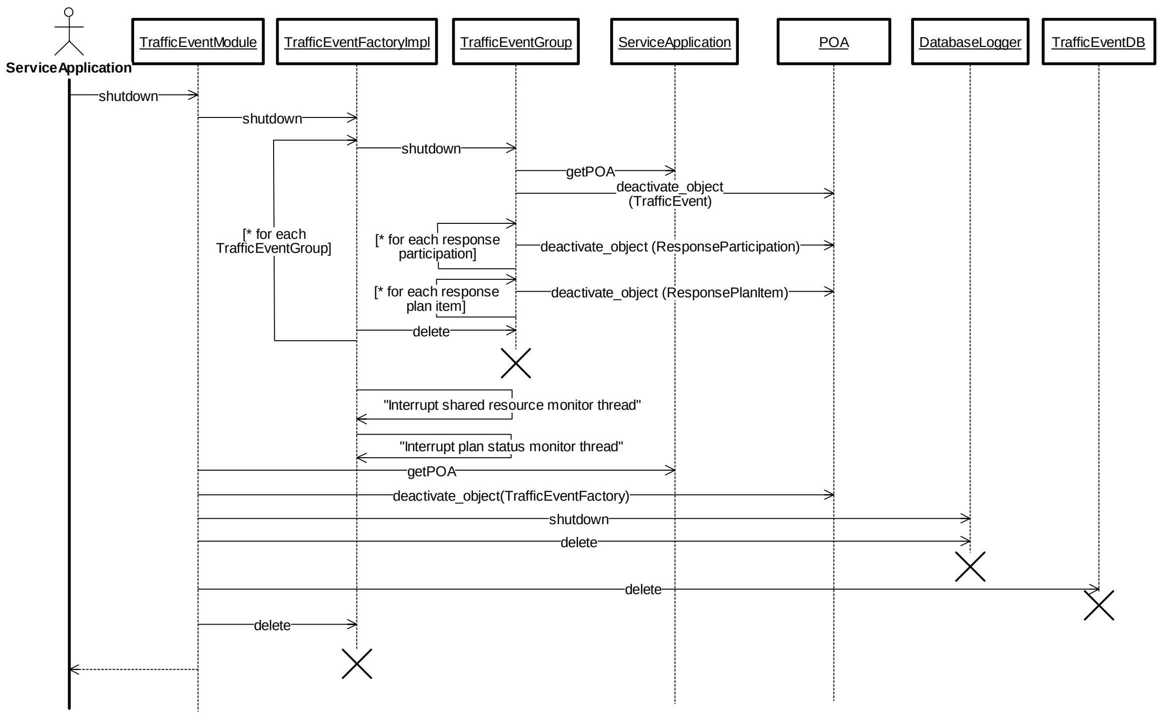

Figure 187. TrafficEventModule:Shutdown (Sequence Diagram)

3.17.2.20 TrafficEventModule:TransferTrafficEvent (Sequence Diagram)

This diagram shows what happens when an event is transferred to another operations center. The TrafficEventImpl is called to set the controlling operations center, and after checking the user's rights, it calls the TrafficEventGroup, which updates the database and calls all of the ResponsePlanItems to tell them that the event has been tranferred. The ResponsePlanItems cause the ArbitrationQueue to be called to transfer the event. A CORBA event is pushed via the event channel and an entry is added to the traffic event's history log.

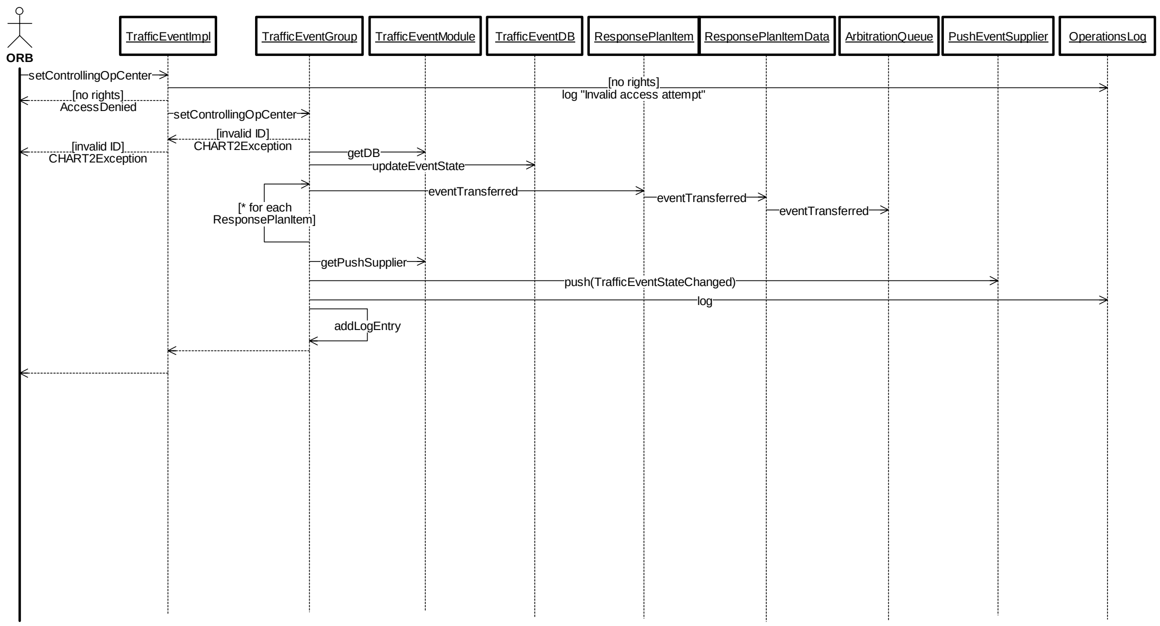

Figure 188. TrafficEventModule:TransferTrafficEvent (Sequence Diagram)

3.18 TTSControl

3.18.1 Classes

3.18.1.1 TTSControlModuleClasses (Class Diagram)

The TTSControlModule serves an instance of the TTSConverter interface, which provides functionality to convert text messages into speech for the CHART2 system. This diagram shows how the implementation of a TTSConverter CORBA interface relies on other supporting classes to perform its functions.

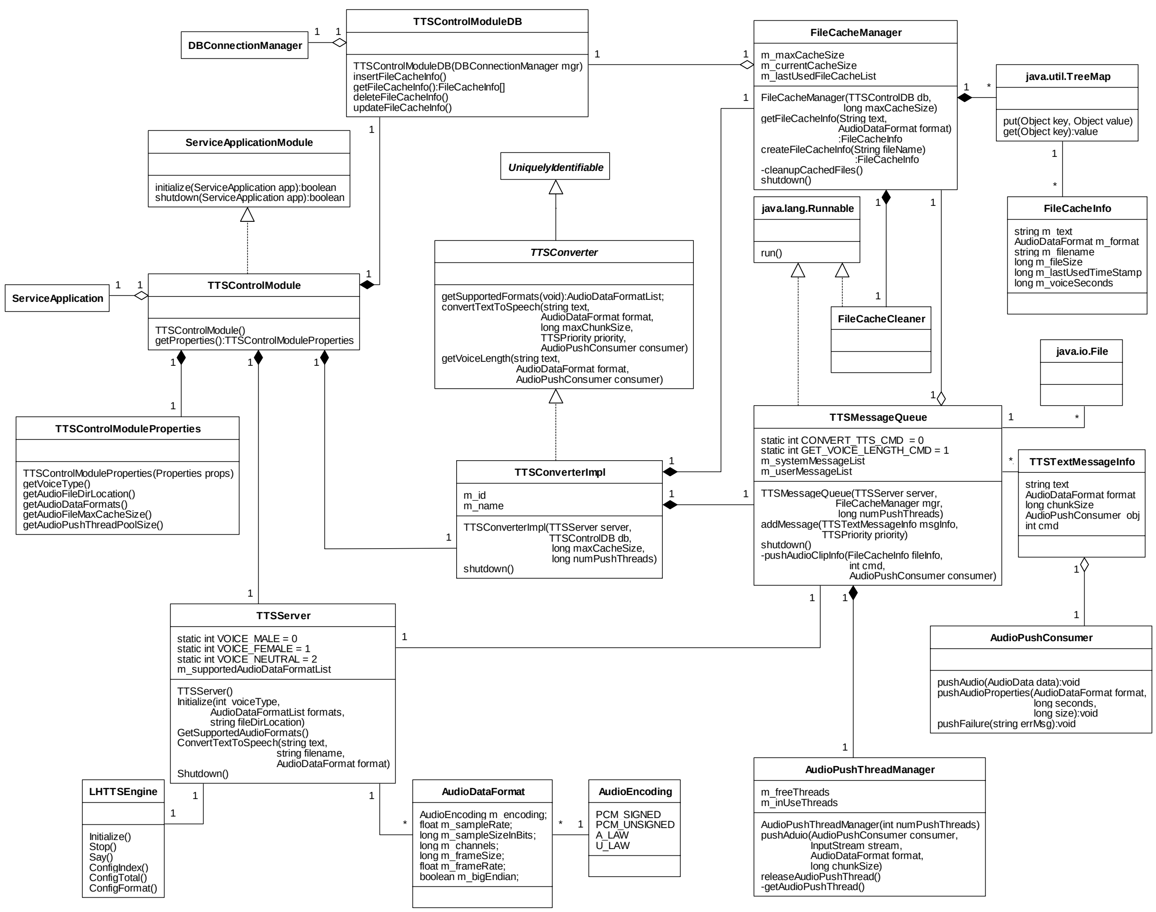

Figure 189. TTSControlModuleClasses (Class Diagram)

3.18.1.1.1 AudioDataFormat (Class)

This structure specifies the format of audio data.

3.18.1.1.2 AudioEncoding (Class)

This enum defines the supported types of encoding for audio data.

3.18.1.1.3 AudioPushConsumer (Class)

This interface is implemented by objects that are intended to receive audio data using the push model, where the server pushes the data to the consumer. One call to pushAudioProperties() will always precede any calls to pushAudio().

3.18.1.1.4 AudioPushThreadManager (Class)

This class maintains a pool of reusable AudioPushThread objects, which can be used to push audio clip information back to the client. It provides the functionality to manage access to the AudioPushThreads.

3.18.1.1.5 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two separate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.18.1.1.6 FileCacheCleaner (Class)

This class represents an instance of a thread that is created to delete the audio clips that have not been used recently when the cache size used by the audio clips exceeds the maximum limit assigned.

3.18.1.1.7 FileCacheInfo (Class)

This structure specifies the information about an audio clip file, which has been converted from a text message to voice and cached for future use.

3.18.1.1.8 FileCacheManager (Class)

This class maintains a mapping between text messages and the corresponding audio clip file information. This is accomplished by maintaining a list of TreeMaps (one for each audio format supported) with text as key and audio clip information as the value. This class also helps manage the amount of hard drive space consumed by the audio clips by deleting the old clip files when the maximum cache size limit is reached. The maximum cache size limit can be set by the administrator using the system properties.

3.18.1.1.9 java.io.File (Class)

This class is an abstract representation of file and directory pathnames.

3.18.1.1.10 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

3.18.1.1.11 java.util.TreeMap (Class)

This class is an implementation of the SortedMap interface. This class guarantees that the map will be in ascending key order, sorted according to the natural order for the key's class, or by the comparator provided at creation time, depending on which constructor is used.

3.18.1.1.12 LHTTSEngine (Class)

This interface represents the L&H RealSpeak Server TTS engine used to convert text messages to speech.

3.18.1.1.13 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.18.1.1.14 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.18.1.1.15 TTSControlModule (Class)

This class implements the Service Application module interface. It publishes the TTSConverterImpl object, which provides the functionality to convert text messages to speech for the CHART2 system. It also creates the RealSpeakServer object, which provides the functionality to access the LHTTSEngine and the TTSControlModuleDB object, which provides access to the database.

3.18.1.1.16 TTSControlModuleDB (Class)

This class is a database accessor class used to store and retrieve audio clip information.

3.18.1.1.17 TTSControlModuleProperties (Class)

This class represents the system properties specific to the TTS Control Module.

3.18.1.1.18 TTSTextMessageInfo (Class)

This struct specifies the text message information required to process text to speech converter request, the call back object to pass the results back and the type of command requested.

3.18.1.1.19 TTSConverter (Class)

This interface represents the Text to Speech converter object that allows text to be passed in and speech to be returned.

3.18.1.1.20 TTSConverterImpl (Class)

This is the implementation of the TTSConverter interface, which provides the functionality to convert text to speech for the CHART2 system.

3.18.1.1.21 TTSMessageQueue (Class)

This class provides the functionality to retrieve messages from the queue and process them by either retrieving the audio clip data using the FileCacheManager object if available or by converting the text messages to speech using the TTSServer object. For text messages not already converted and available in the cache, this class maintains two queues of messages to be converted into speech, one for message requests from the system and another for the users. The messages in system message queue get a higher priority over messages in user message queue. All the messages of a particular queue are processed in a First In First Out fashion. The audio data produced from conversion or retrieved from the cache is passed back to the client via the AudioPushConsumer object using the AudioPushThreadManager object.

3.18.1.1.22 TTSServer (Class)

This class provides the functionality to access and control the TTS Engine from the CHART2 system. It provides the functionality to start, stop and change the configuration of the TTS Engine. It also provides a method to convert a text message to speech.

3.18.1.1.23 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.18.2 Sequence Diagrams

3.18.2.1 TTSControlModule:AddMessageToQueue (Sequence Diagram)

This diagram shows how a TTSConverter request is added to the message queue. First, the TTSMessageQueue queries the FileCacheManager to check if an already converted audio clip exists for the text message of the desired audio format. The FileCacheManager looks in the TreeMap of the desired audio format for the audio clip using the text message as the key. The TreeMap returns the audio clip file information, if the audio clip already exists. Otherwise, it returns a null. If the audio clip was not found, the message is queued in the proper queue depending upon the priority and the request returns (see ProcessQueuedMessages sequence digram for details about how the queued messages are processed). If the audio clip is found, the last used timestamp in the file cache information is updated and the audio data is pushed back to caller using the AudioPushConsumer object passed with the request (see PushAudioClipInformation for details about how audio clip data are passed back to the client).

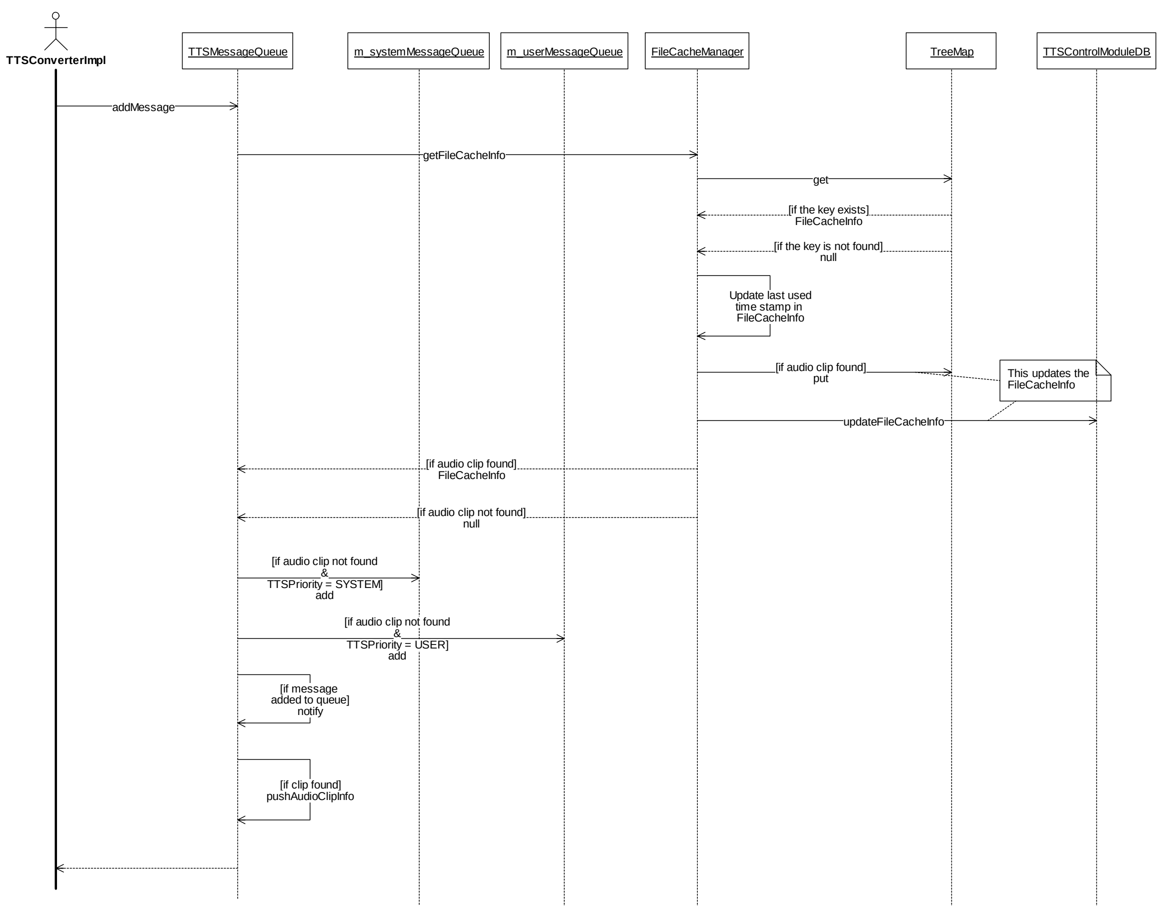

Figure 190. TTSControlModule:AddMessageToQueue (Sequence Diagram)

3.18.2.2 TTSControlModule:CleanupFileCache (Sequence Diagram)

This diagram shows how the FileCacheManager thread deletes the old audio clip files when the cache limit is exceeded.

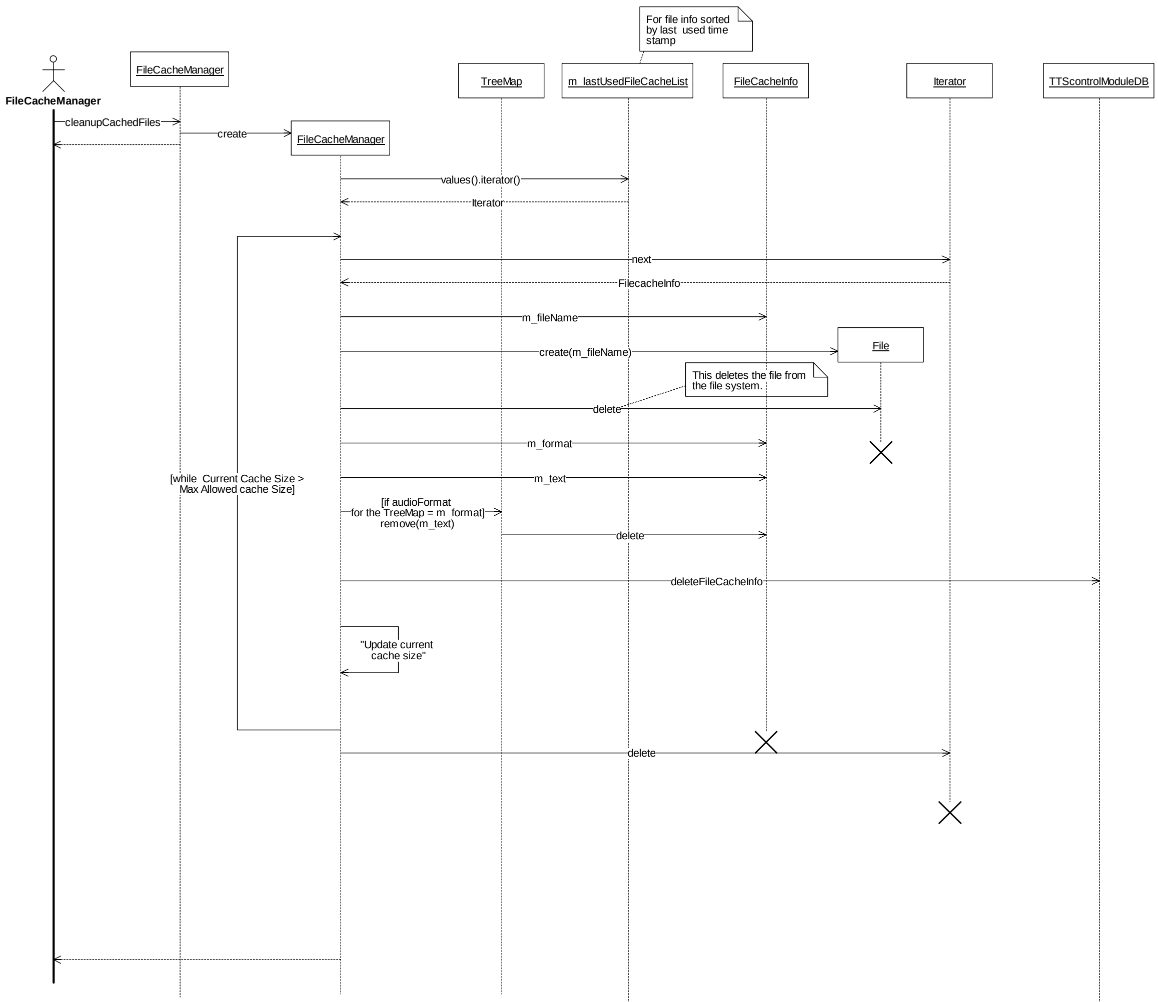

Figure 191. TTSControlModule:CleanupFileCache (Sequence Diagram)

3.18.2.3 TTSControlModule:ConvertTextToSpeech (Sequence Diagram)

This sequence diagram shows how a convert text to speech request is processed. The message is added to the TTSMessageQueue and audio clip information will be pushed back using the AudioPushComsumer object passed through this call. See ProcessQueuedMessages and HARUtility.PushAudio sequence diagrams for details about how the messages are processed and the data is pushed back.

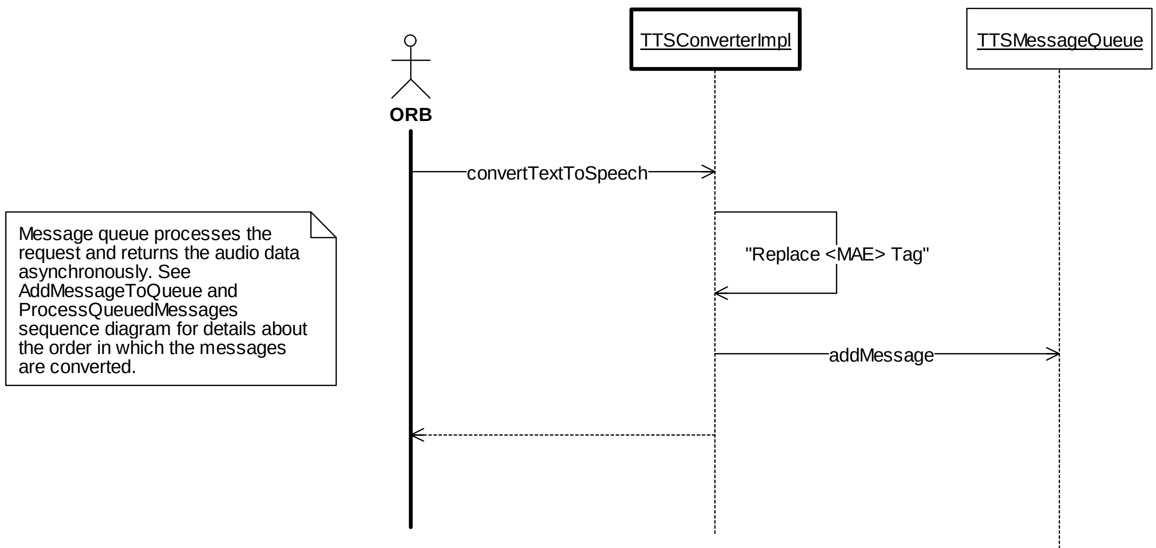

Figure 192. TTSControlModule:ConvertTextToSpeech (Sequence Diagram)

3.18.2.4 TTSControlModule:CreateFileCacheInfo (Sequence Diagram)

This diagram shows how the FileCacheManager creates a FileCacheInfo object, which stores the text message and audio clip file information for future use. A file object is created from the given file name and is passed to AudioSystem class to get the AudioInputStream object, which contains the audio format information and the actual data. The length of the audio message and the size of the audio file are calculated using the audio format properties. The AudioDataFormat object is created and a FileCacheInfo object is created using the various data available. Finally, the FileCacheInfo object is added to the TreeMap containing others clip information of similar audio format and the FileCacheInfo object is returned.

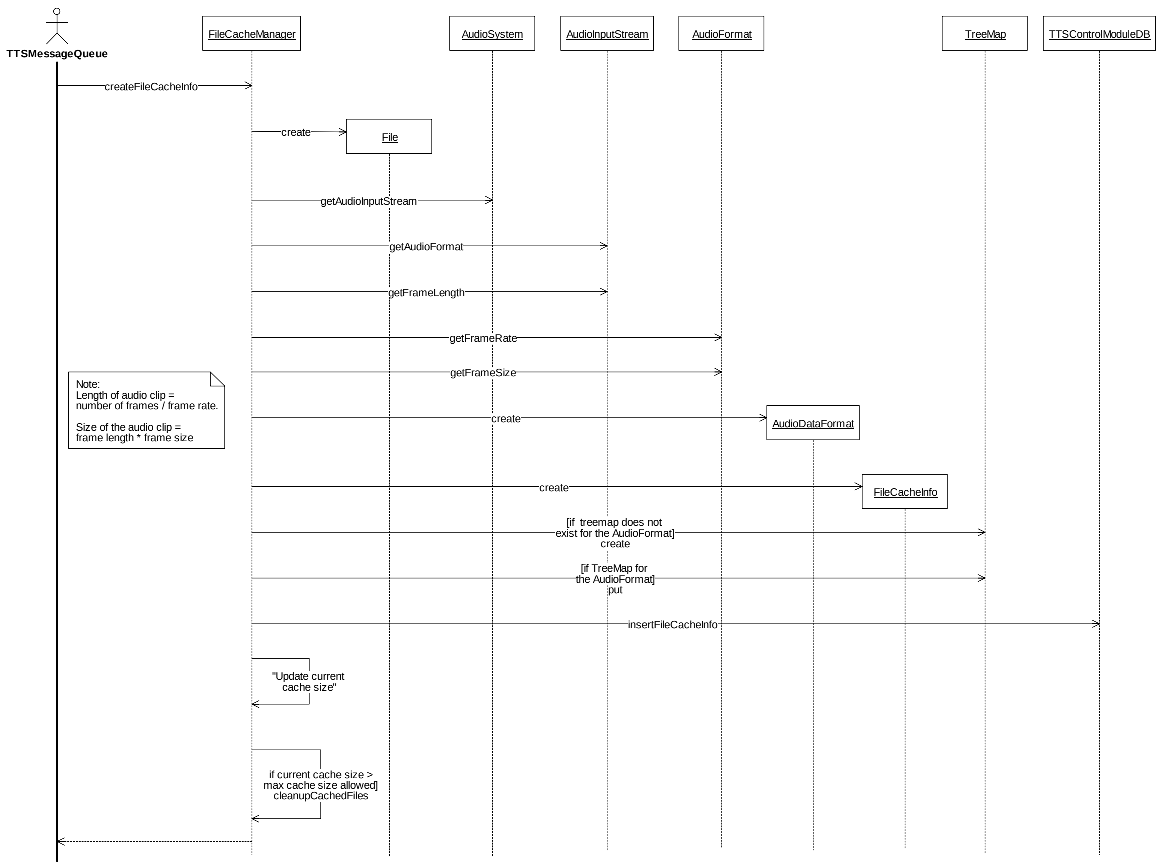

Figure 193. TTSControlModule:CreateFileCacheInfo (Sequence Diagram)

3.18.2.5 TTSControlModule:GetSupportedFormats (Sequence Diagram)

This diagram shows how to retrieve a list of currently supported audio formats from the TTS Engine.

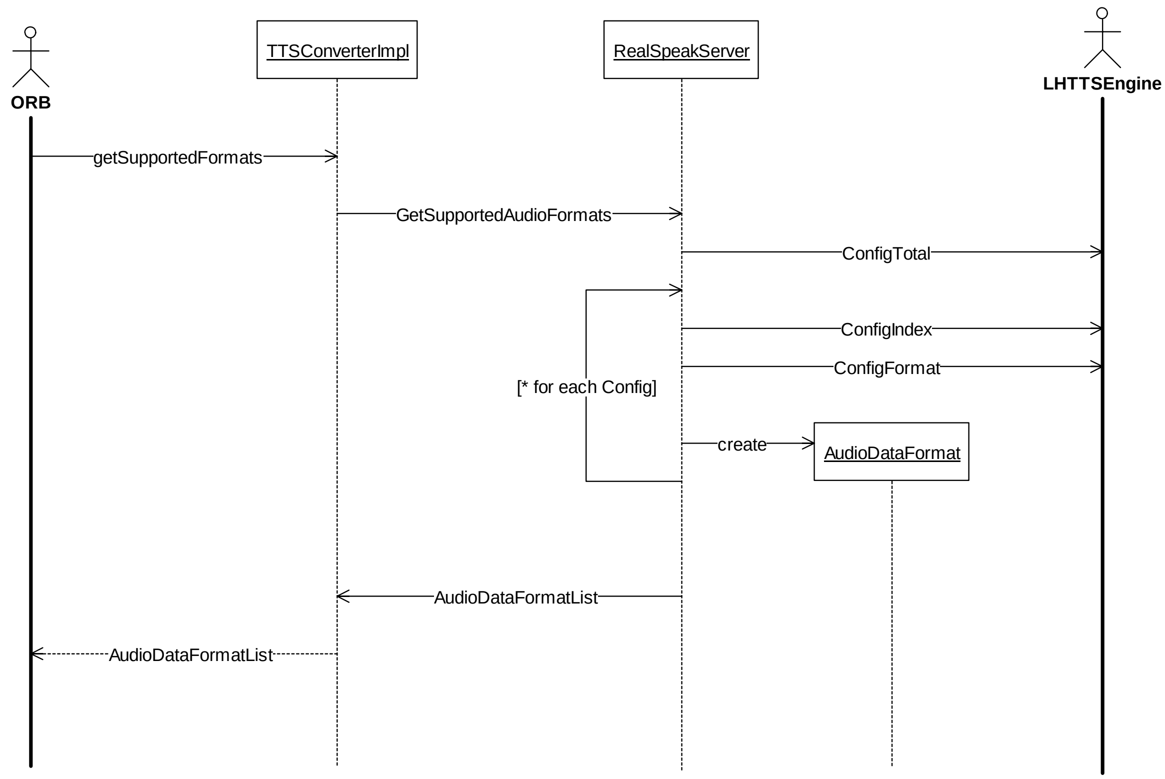

Figure 194. TTSControlModule:GetSupportedFormats (Sequence Diagram)

3.18.2.6 TTSControlModule:Initialize (Sequence Diagram)

This diagram shows the sequence of operations that takes place when the TTSControlModule is initialized. Upon creation, the TTSControlModule creates a TTSControlServiceProperties object, which provides the user defined system properties to the rest of the objects in the TTSControlModule. A TTSControlDB object is created to provide access to the database for TTSControlModule. A TTSServer object is created to control and provide access to the TTS engine. A TTSConverterImpl object is created, activated with the POA and published in the Trader to provide the capability to convert text to speech for the rest of the CHART2 system. The TTSConverterImpl object creates a TTSMessageQueue thread, which provides the functionality to queue and prioritize the TTSConverter requests. The TTSConverterImpl object also creates a FileCacheManager object, which manages the audio clip file info. The TTSMessageQueue creates an AudioPushThreadManager object, which contains a pool of AudioPushThreads that can be used to push audio clip information back to the clients of the TTSConverter. The number of AudioPushThreads to be created can be configured through the system properties file.

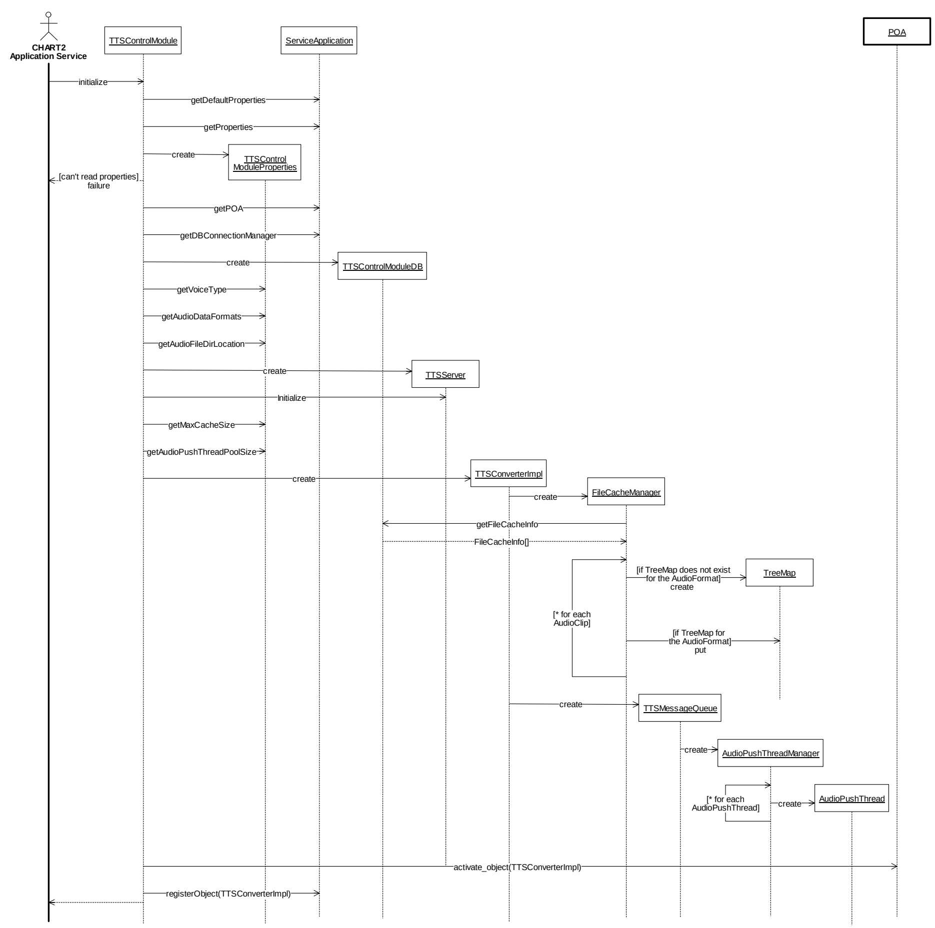

Figure 195. TTSControlModule:Initialize (Sequence Diagram)

3.18.2.7 TTSControlModule:GetVoiceLength (Sequence Diagram)

This sequence diagram shows how a request to get audio message length is processed. The message is added to the TTSMessageQueue and audio clip information will be pushed back using the AudioPushComsumer object passed through this call. See ProcessQueuedMessages sequence diagrams for details about how the messages are processed and the data is pushed back.

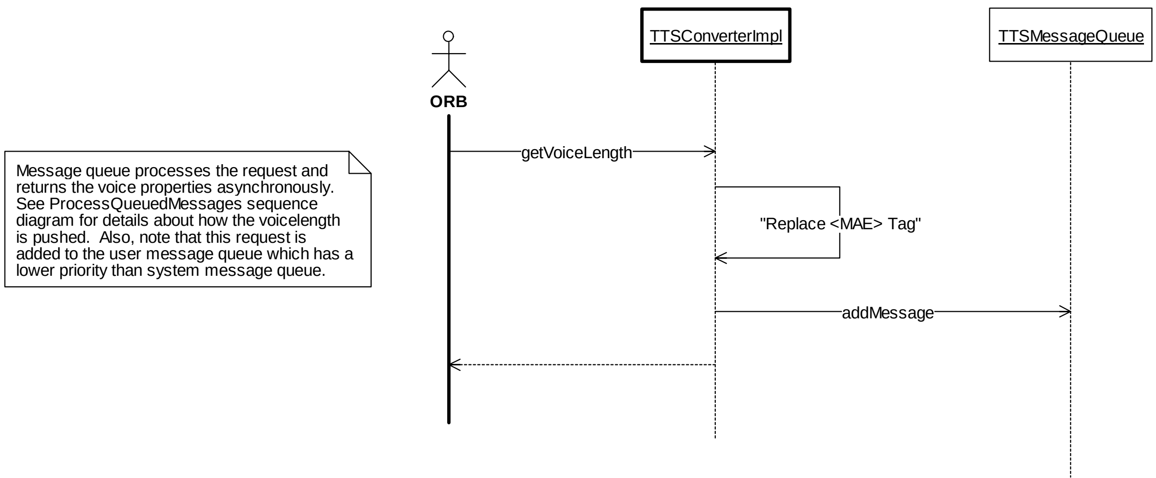

Figure 196. TTSControlModule:GetVoiceLength (Sequence Diagram)

3.18.2.8 TTSControlModule:ProcessQueuedMessages (Sequence Diagram)

This diagram shows how TTSMessageQueue thread processes the queued messages. The thread continuously looks for messages added to System Message Queue and User Message Queue. At any time, messages queued in the System Message Queue have a higher priority over the messages queued in the User Message Queue. Once a message is retrieved from the queue, a check is made to see if the same text message with the desired audio format has been converted before. If the audio clip file is found, the audio data is pushed back to client using the AudioPushConsumer object passed with the request. If a pre-converted clip is not available, the thread requests the TTSServer to convert the text message to speech. If the TTS engine fails to convert the message, the consumer is notified. If the message is converted successfully, the audio clip information is stored in the FileCacheManager for future use and the audio properties are pushed to the client. See PushAudioClipInfo sequence diagram for details about how the audio clip information is pushed.

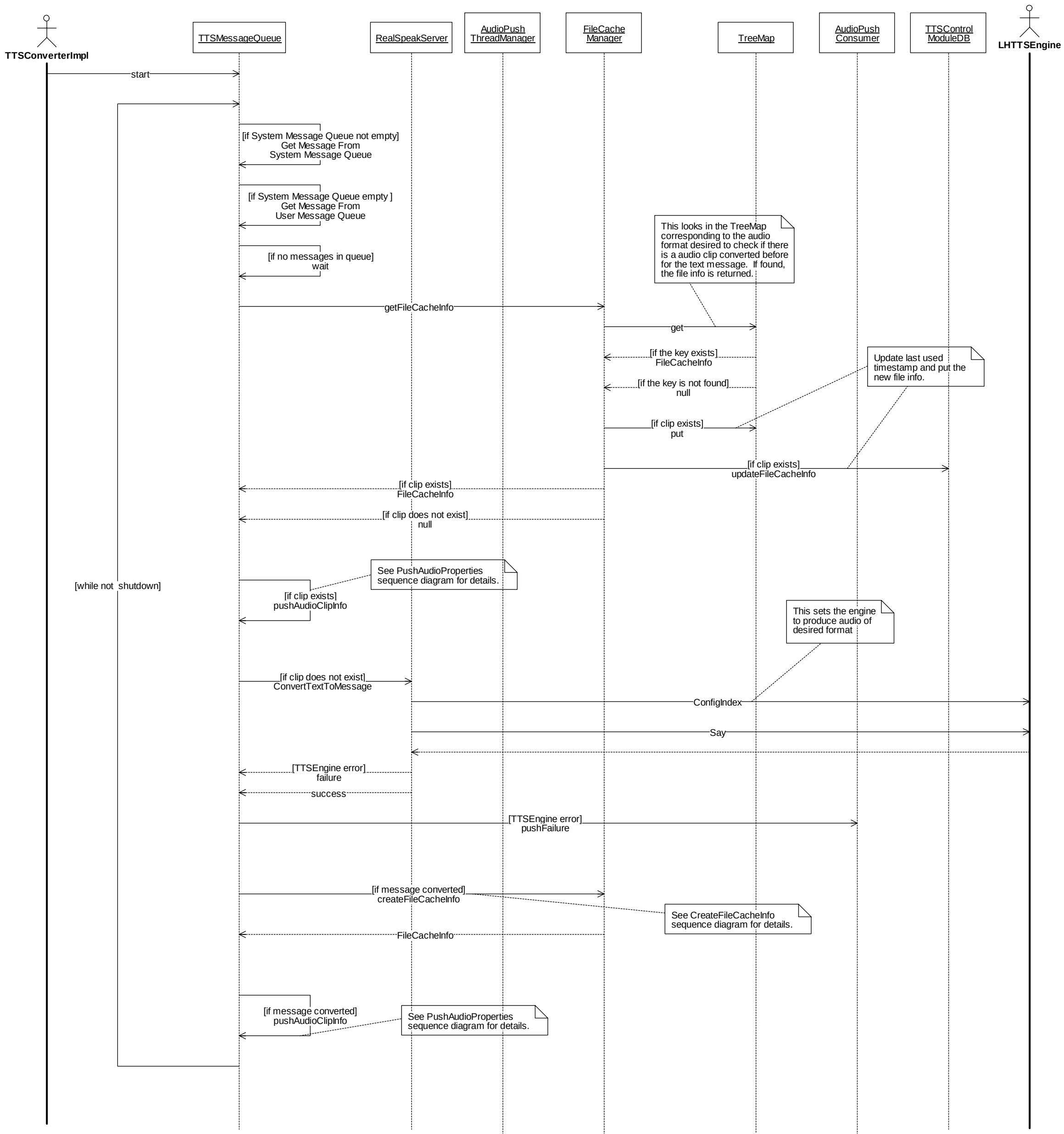

Figure 197. TTSControlModule:ProcessQueuedMessages (Sequence Diagram)

3.18.2.9 TTSControlModule:PushAudioClipInformation (Sequence Diagram)

This diagram shows how the audio clip information is pushed back to the caller of a TTSConverter request. If the request is a get voice length command, the audio clip properties are pushed to client using the AudioPushConsumer passed with the request. If the request is for converted audio data, a File object is created to access the audio to retrieve the audio data. An AudioInputStream object is retrieved using the AudioSytem class. The input stream along with the AudioPushConsumer is passed to the AudioPushThreadManager for pushing the audio data. See HARUtility.PushAudio for details about how the audio data is pushed.

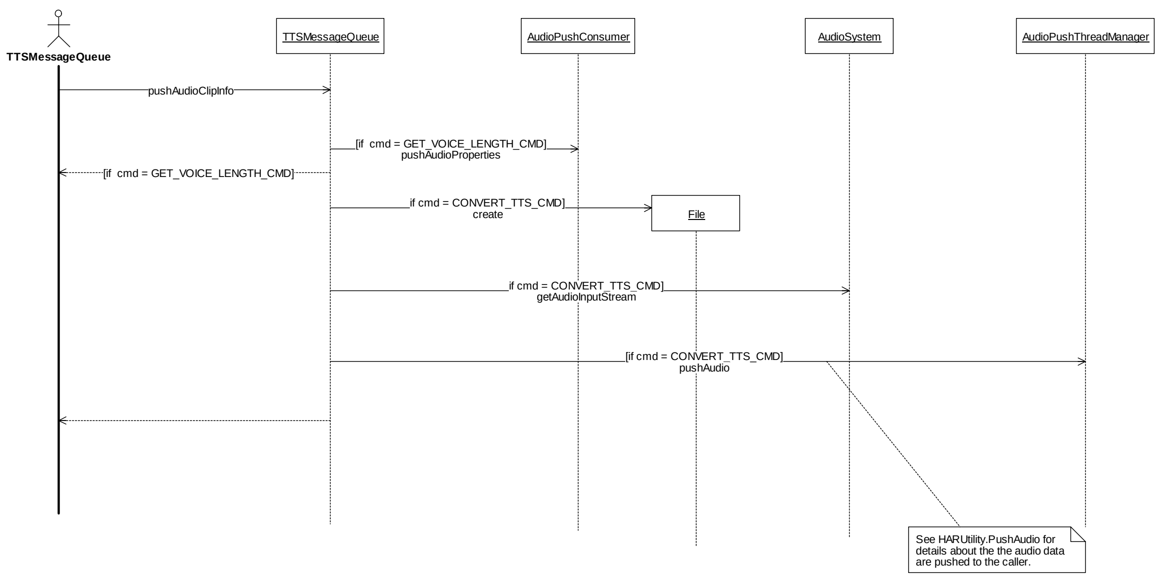

Figure 198. TTSControlModule:PushAudioClipInformation (Sequence Diagram)

3.18.2.10 TTSControlModule:Shutdown (Sequence Diagram)

This diagram shows the sequence of operations that takes place when the TTSControlModule is shutdown. The TTSConverterImpl object is deactivated and shutdown. The TTSConverterImpl object in turn shuts down the TTSMessageQueue thread, which causes to shutdown the AudioPushThreadManager thread and AudioPushThreads. The TTSServer object is also shutdown and the TTSControlDB object is destroyed.

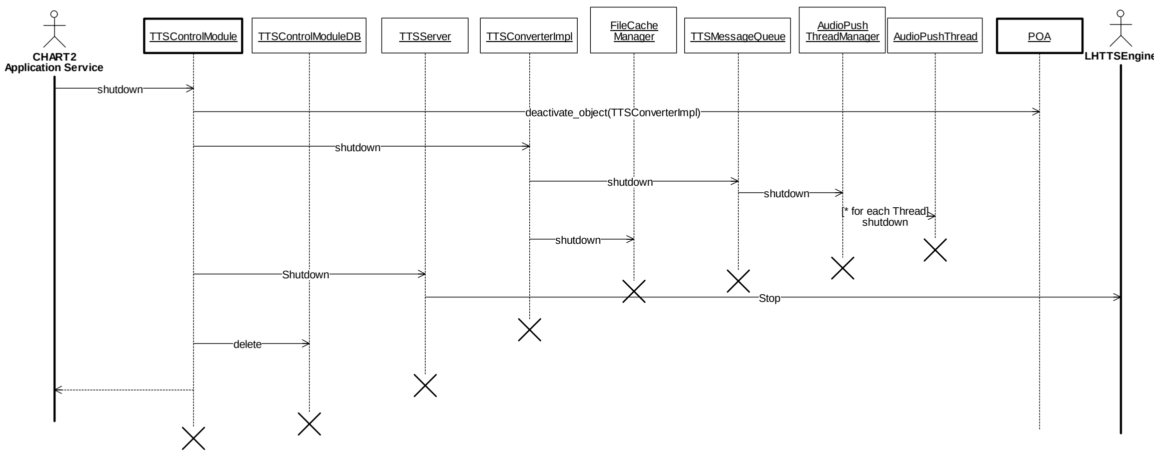

Figure 199. TTSControlModule:Shutdown (Sequence Diagram)

3.19 UserManagementModule

3.19.1 Classes

3.19.1.1 UserManagementModuleClasses (Class Diagram)

This class diagram shows classes that support user management in the CHART II system. The purpose of this module is to serve the object implementing the UserManager interface and to serve the objects implementing the Profile interface.

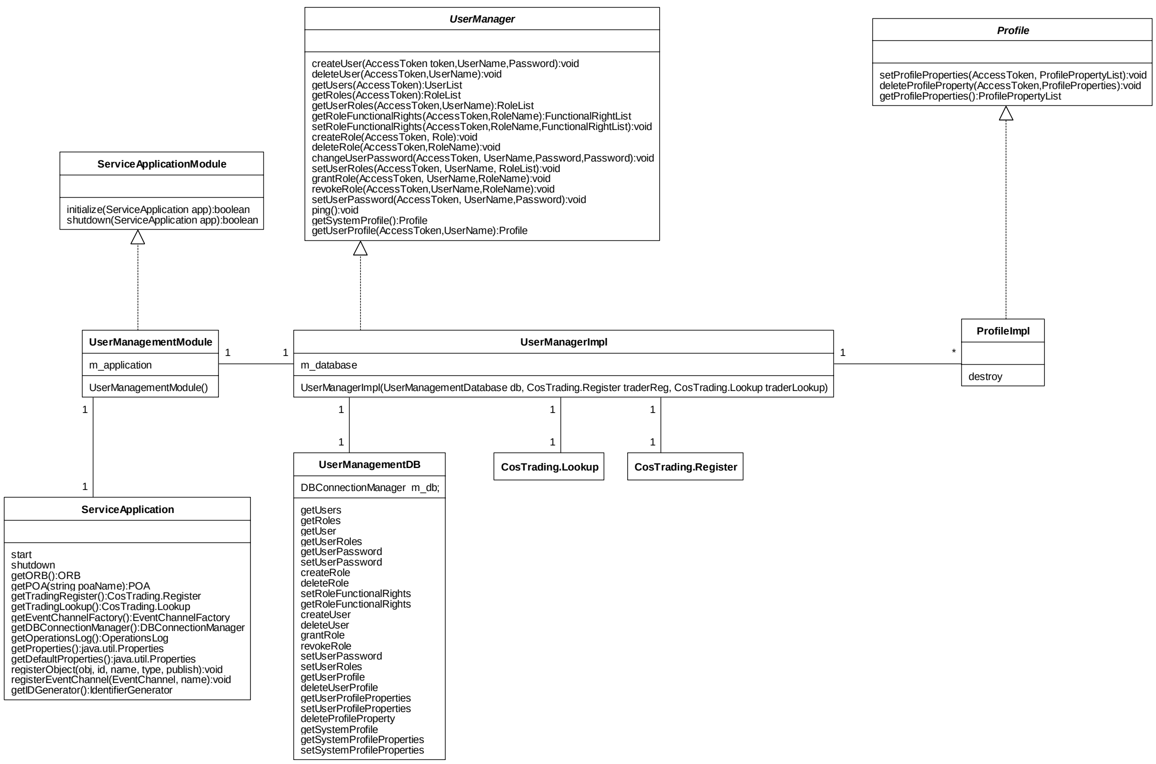

Figure 200. UserManagementModuleClasses (Class Diagram)

3.19.1.1.1 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects that have previously been published.

3.19.1.1.2 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

3.19.1.1.3 Profile (Class)

This class contains a set of user or administrator defined properties that are used to configure how the CHART II system behaves or presents information to a user.

3.19.1.1.4 ProfileImpl (Class)

This class is the specific implementation of a Profile interface that will be served by the User Management Service. As such, it contains the profile properties and provides methods to get, add and delete the properties.

3.19.1.1.5 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.19.1.1.6 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.19.1.1.7 UserManagementDB (Class)

The UserManagementDB Class provides methods used to access and modify User Managment data in the database. This class uses a Database object to retrieve a connection to the database for its exclusive use during a method call.

3.19.1.1.8 UserManagementModule (Class)

This module creates, publishes and deletes the object that implements the UserManager interface for user configuration and manipulation.

3.19.1.1.9 UserManager (Class)

The UserManager provides access to data dealing with user management. This includes users, roles, and functional rights. The UserManager is largely an interface to the User Management database tables.

3.19.1.1.10 UserManagerImpl (Class)

This class is the specific implementation of a UserManager interface that will be served by the User Management Service. As such, it provides implementations of each of the methods in the UserManger interface.

3.19.2 Sequence Diagrams

3.19.2.1 UserManagementModule:AddUser (Sequence Diagram)

A user with the proper functional rights may add a new user to the system. The user will be added to the user database provided the password and username specified for the new user are valid.

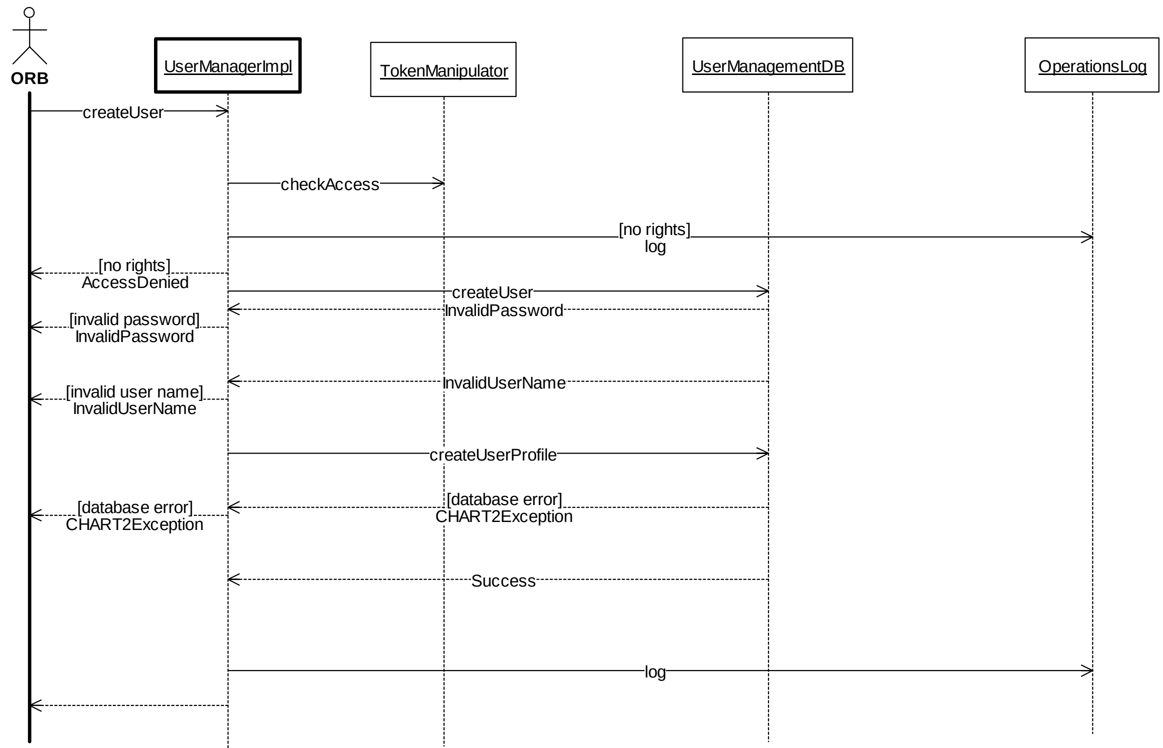

Figure 201. UserManagementModule:AddUser (Sequence Diagram)

3.19.2.2 UserManagementModule:ChangeUserPassword (Sequence Diagram)

A user may change his/her own password. The system will verify that the invoking user is actually the user whose password is being changed and will require the user to pass his/her current password that must match the password in the user database.

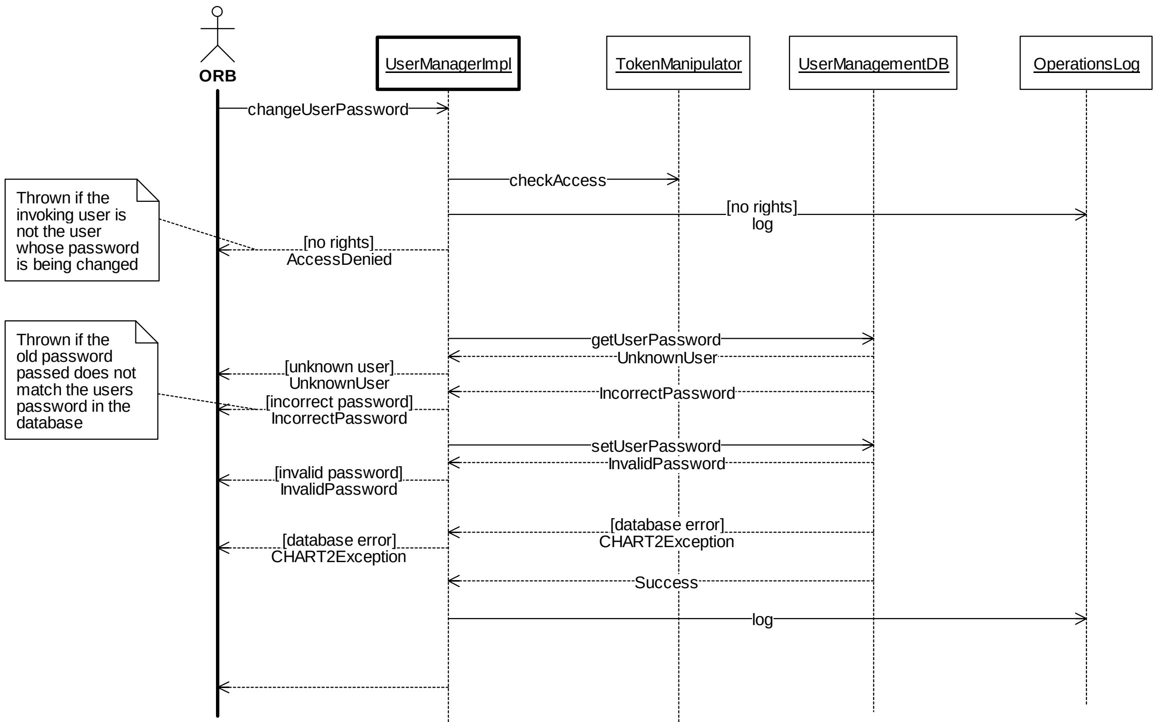

Figure 202. UserManagementModule:ChangeUserPassword (Sequence Diagram)

3.19.2.3 UserManagementModule:CreateRole (Sequence Diagram)

A user with the proper functional rights may create a new role in the user database. The system will verify that the role is not already defined before creating it.

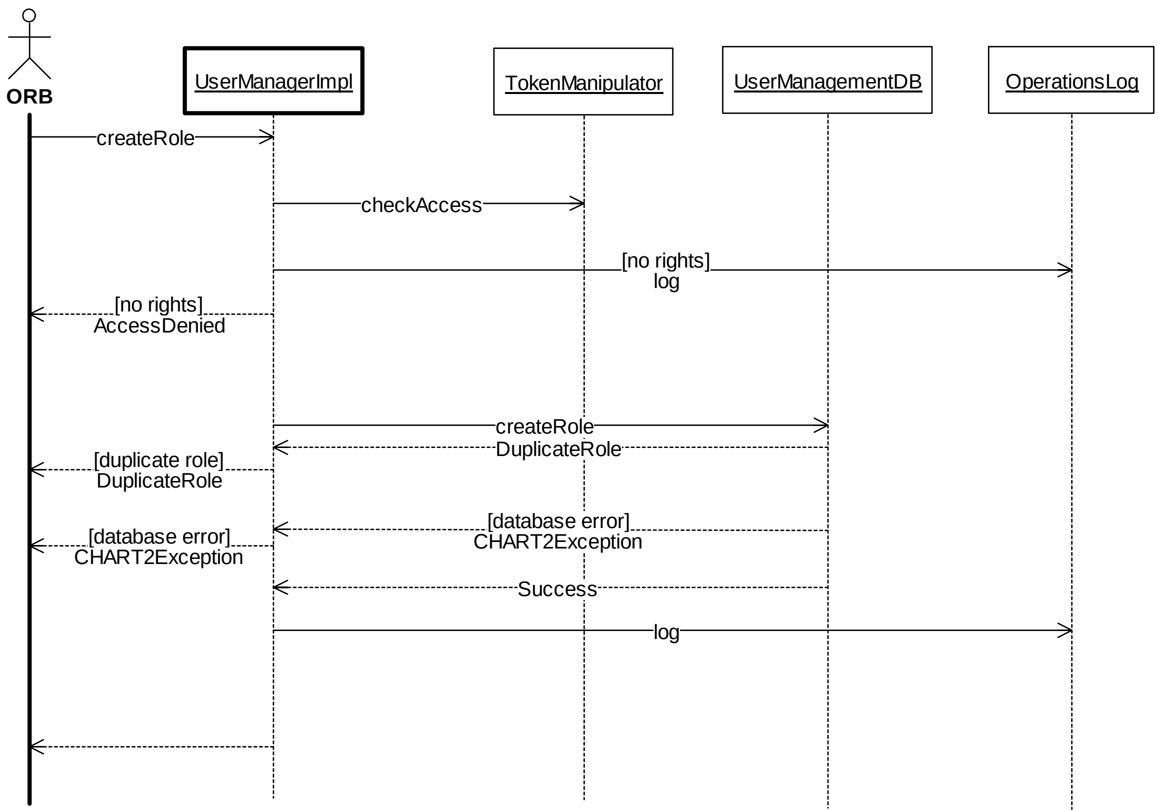

Figure 203. UserManagementModule:CreateRole (Sequence Diagram)

3.19.2.4 UserManagementModule:DeleteProfileProperty (Sequence Diagram)

A user with proper functional rights can delete a profile property from the profile.

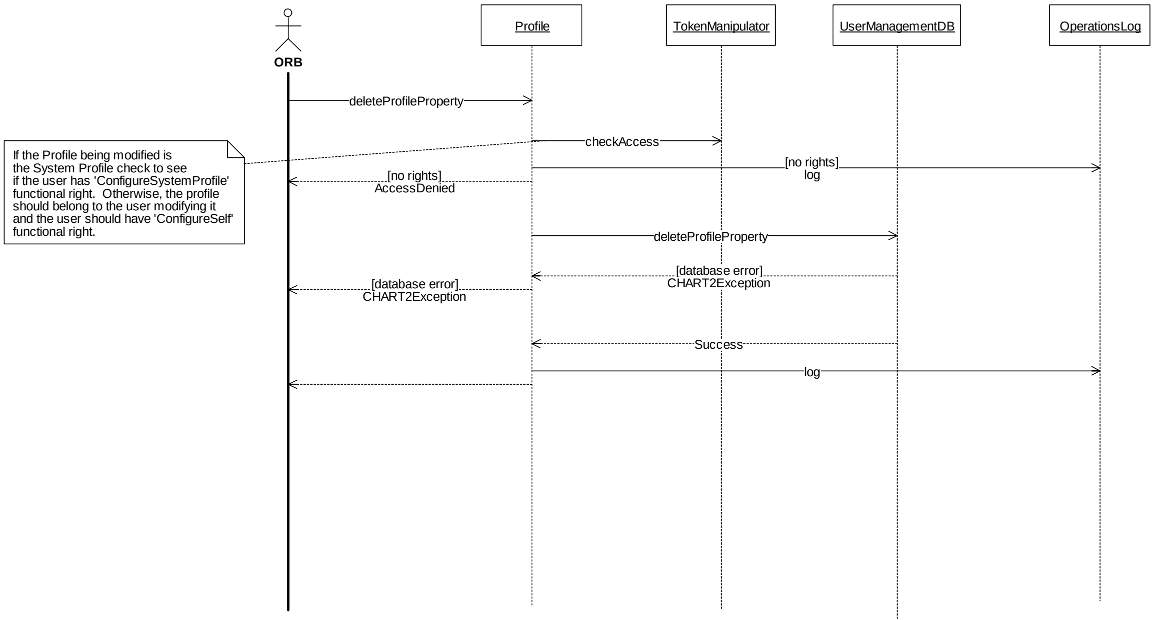

Figure 204. UserManagementModule:DeleteProfileProperty (Sequence Diagram)

3.19.2.5 UserManagementModule:DeleteRole (Sequence Diagram)

A user with the proper functional rights may delete a role from the user database. The system will verify that the role is not currently assigned to any users before deleting it.

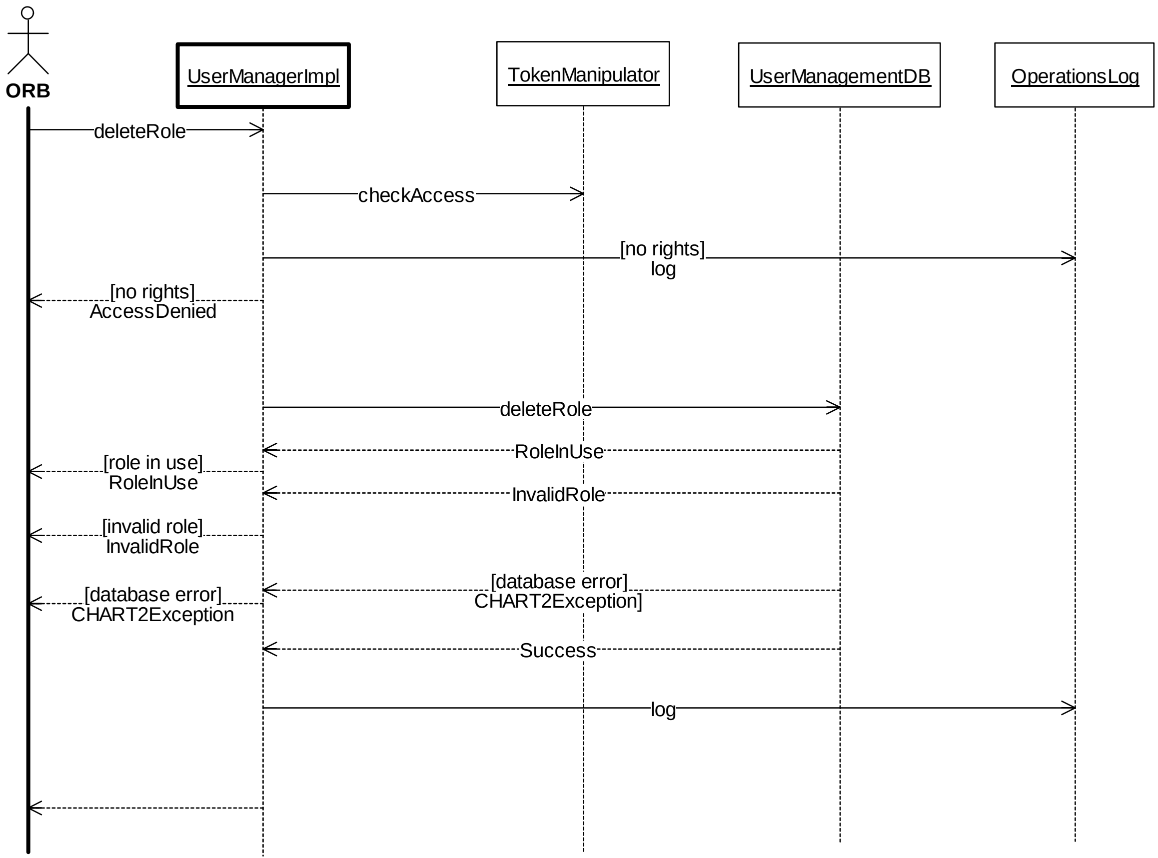

Figure 205. UserManagementModule:DeleteRole (Sequence Diagram)

3.19.2.6 UserManagementModule:DeleteUser (Sequence Diagram)

A user with the proper functional rights may delete a user from the user database. The system will check if the user who is being deleted is currently logged in. If the user is logged in, the administrator will be notified of this fact and will not be able to delete the user. Note that the administrator may use the system to force the user to logout and then delete the user. The check to see if the user is currently logged in is a warning to the administrator and, due to its use of the trader, cannot be guaranteed to successfully check all logins. If the user is deleted from the database while logged in, however, it will not affect his/her current session. He/she will simply not be able to use the system subsequent to logging out.

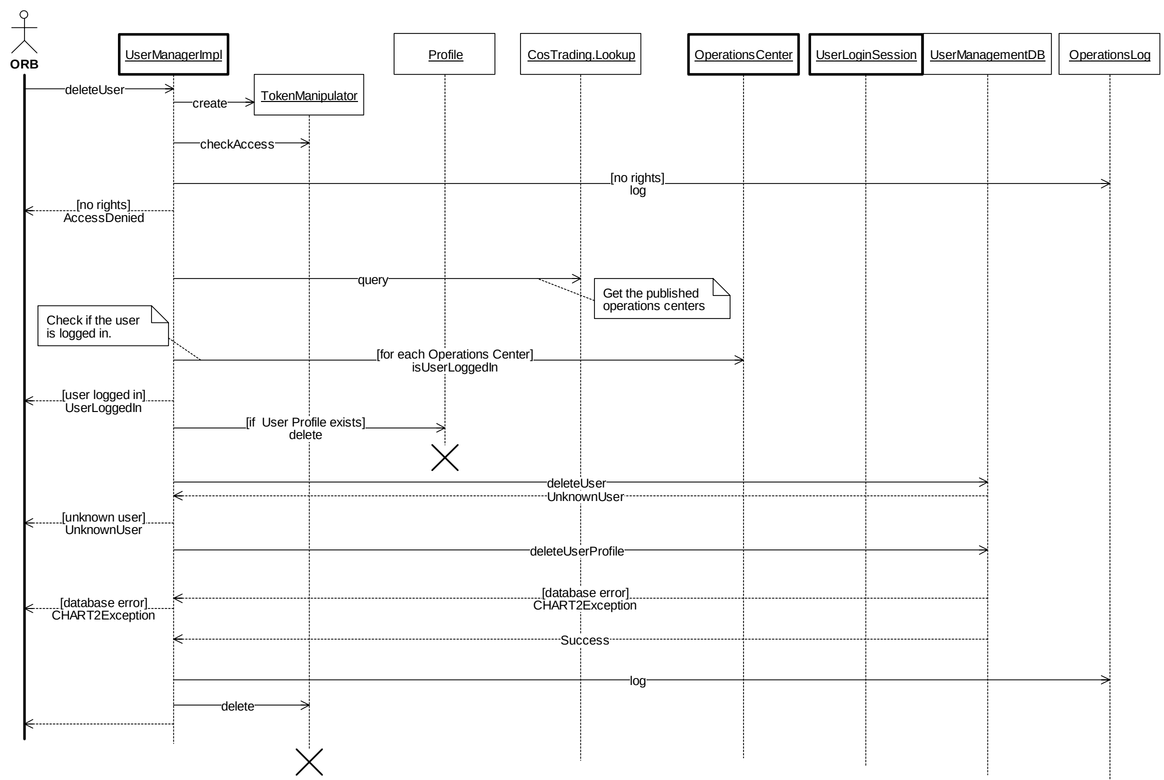

Figure 206. UserManagementModule:DeleteUser (Sequence Diagram)

3.19.2.7 UserManagementModule:GetSystemProfile (Sequence Diagram)

A user can get the system profile that is common to all the users in the CHART2 system.

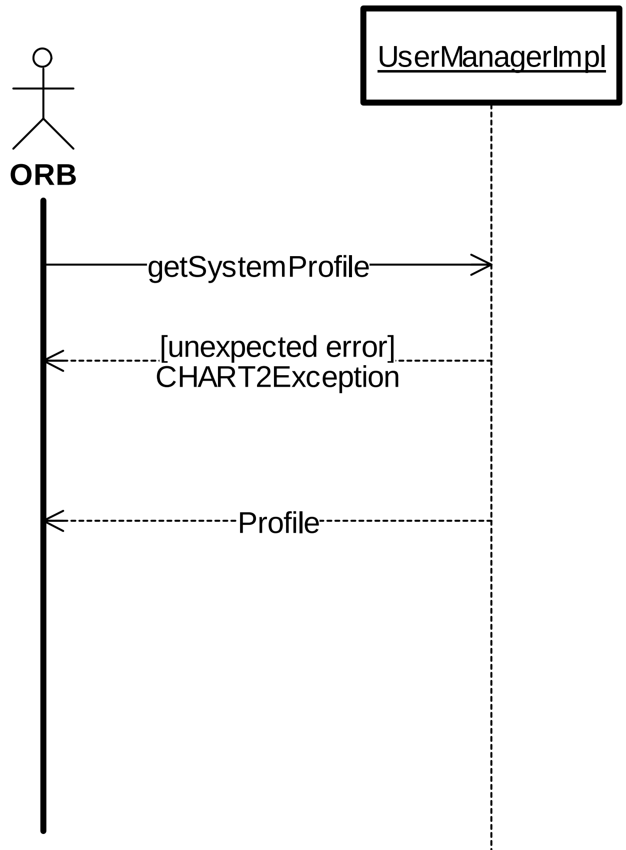

Figure 207. UserManagementModule:GetSystemProfile (Sequence Diagram)

3.19.2.8 UserManagementModule:GetUserProfile (Sequence Diagram)

A user with proper functional rights can get his or her own Profile.

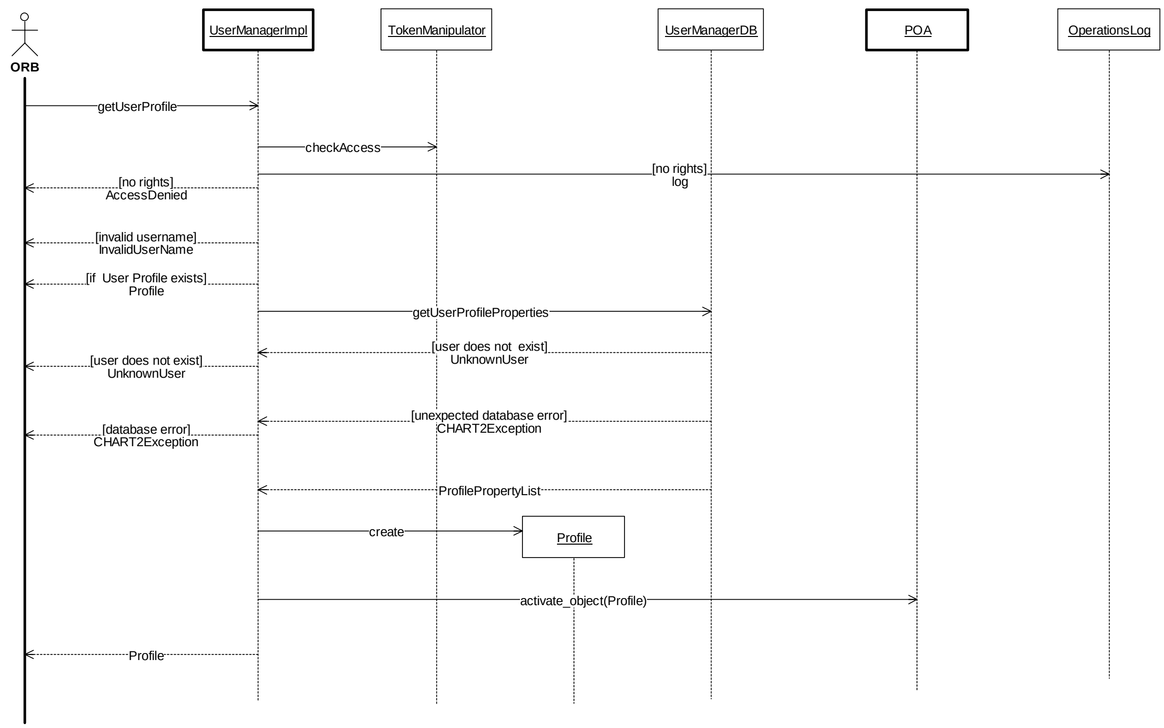

Figure 208. UserManagementModule:GetUserProfile (Sequence Diagram)

3.19.2.9 UserManagementModule:GrantRole (Sequence Diagram)

A user with the proper functional rights may grant a role to a user. The user will not get his/her new functional rights until he/she logs off and logs back on.

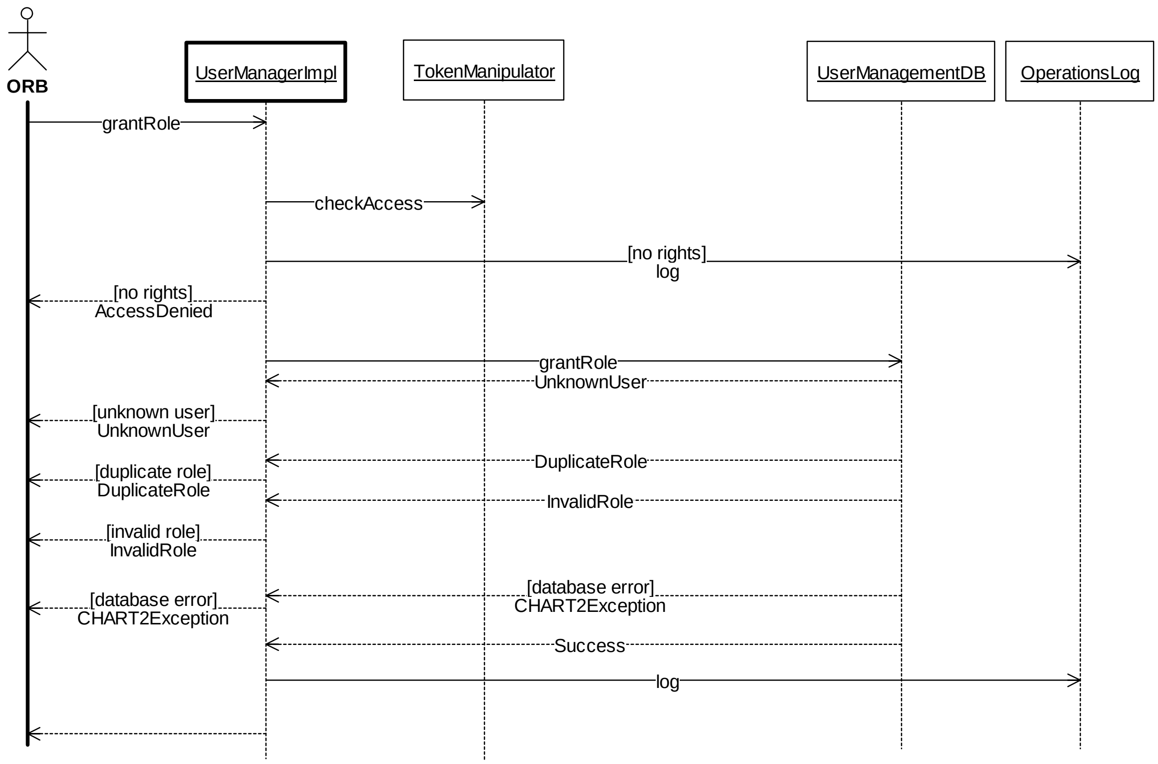

Figure 209. UserManagementModule:GrantRole (Sequence Diagram)

3.19.2.10 UserManagementModule:Initialize (Sequence Diagram)

Upon initialization the user manager module will create the objects which it is responsible for serving, activates them using the POA, and exports them to the CORBA trading service. After initialization this module is available for use by clients.

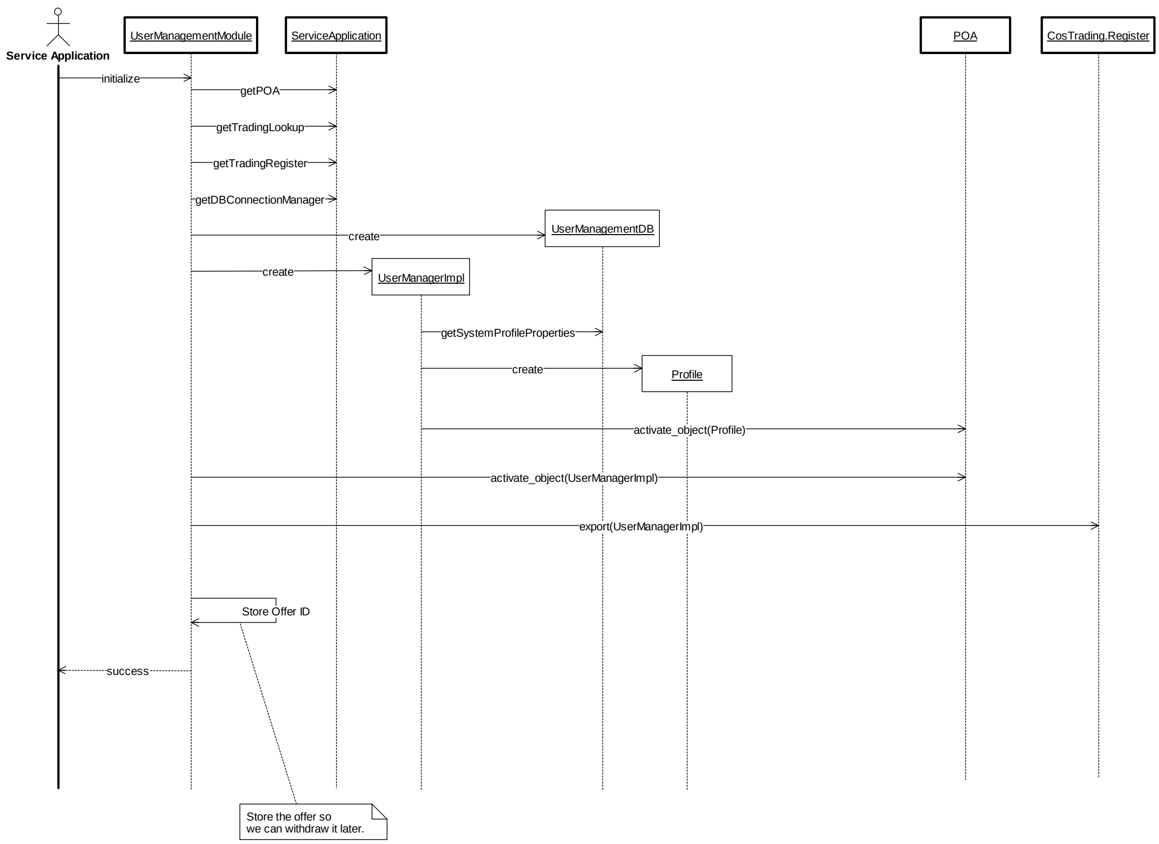

Figure 210. UserManagementModule:Initialize (Sequence Diagram)

3.19.2.11 UserManagementModule:ModifyRole (Sequence Diagram)

A user with the proper functional rights may change the functional rights that belong to a role. This will have the effect of changing the actions that users who have been granted that role may perform. However, these changes will not be recognized until the user logs out and logs back in.

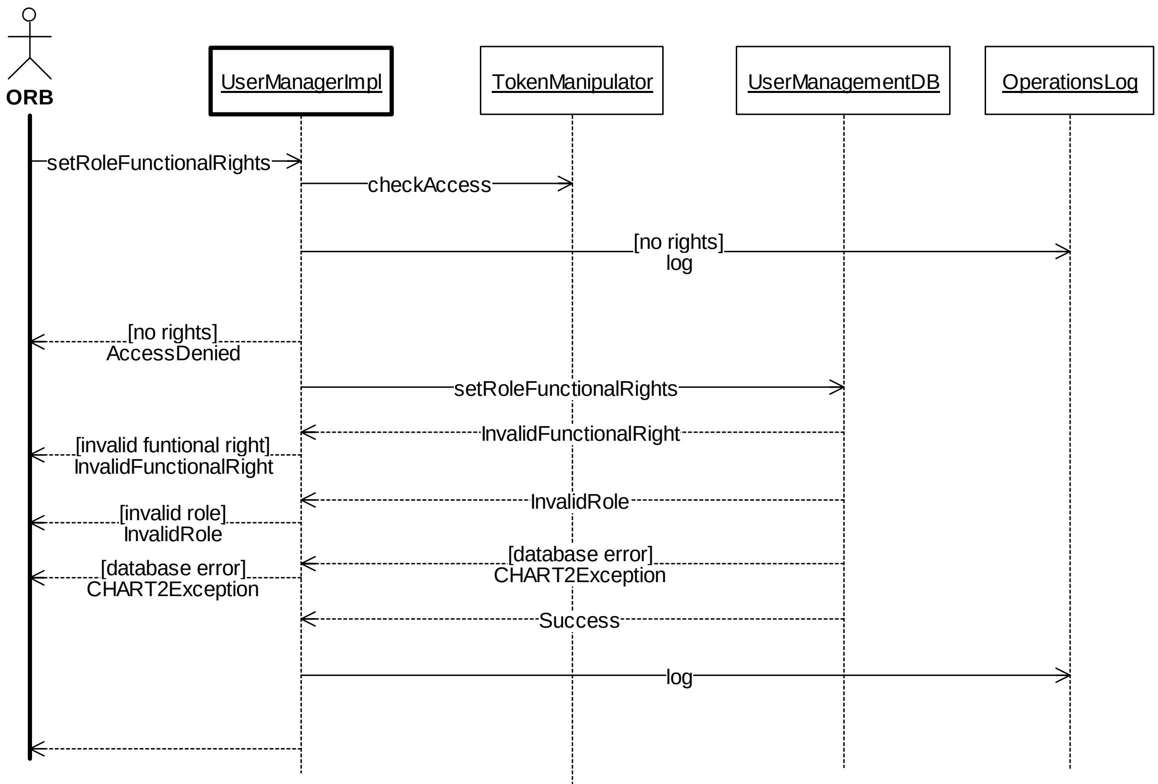

Figure 211. UserManagementModule:ModifyRole (Sequence Diagram)

3.19.2.12 UserManagementModule:RevokeRole (Sequence Diagram)

A user with the proper functional rights may revoke a role that has previously been granted to a user. This action will result in the user having a reduced set of functional rights, and thus reduce the number of system activities the user may perform. The user will get his/her new list of functional rights the next time he/she logs in.

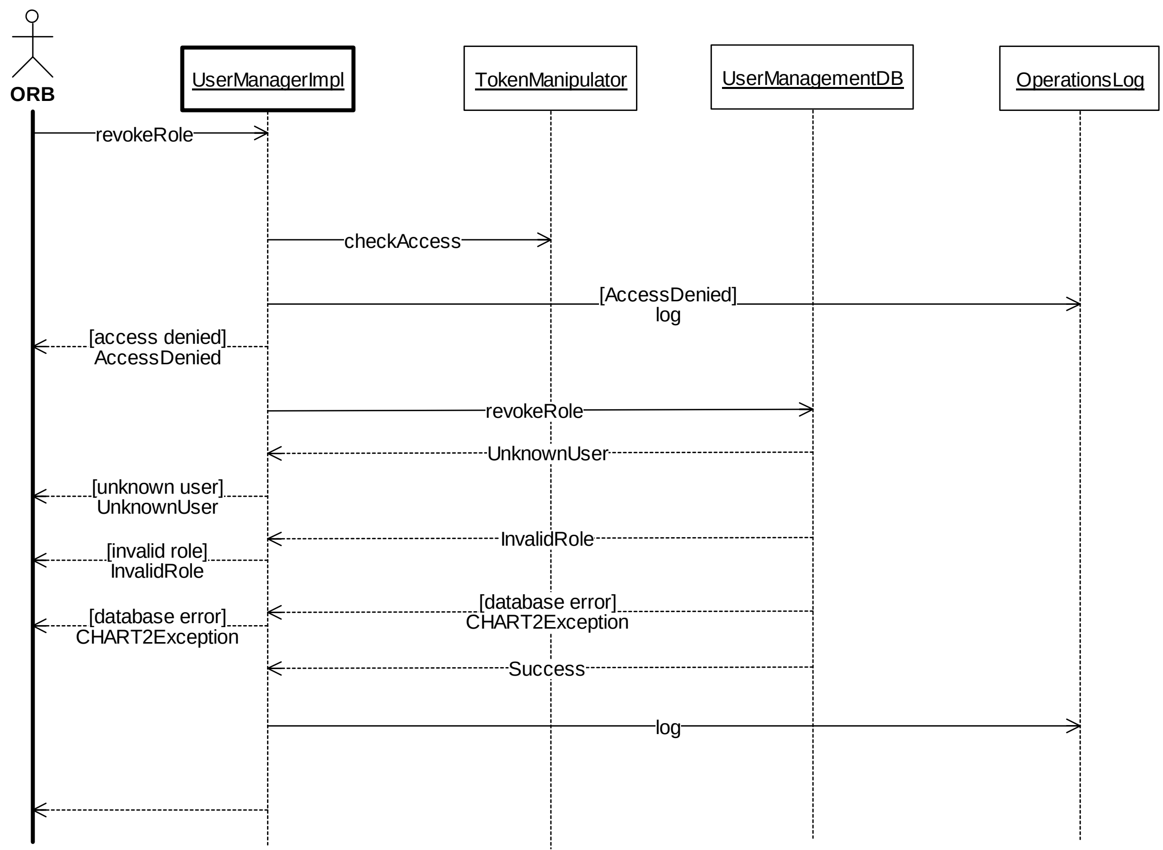

Figure 212. UserManagementModule:RevokeRole (Sequence Diagram)

3.19.2.13 UserManagementModule:SetProfileProperties (Sequence Diagram)

A user with the proper functional rights can store a set of properties in a profile.

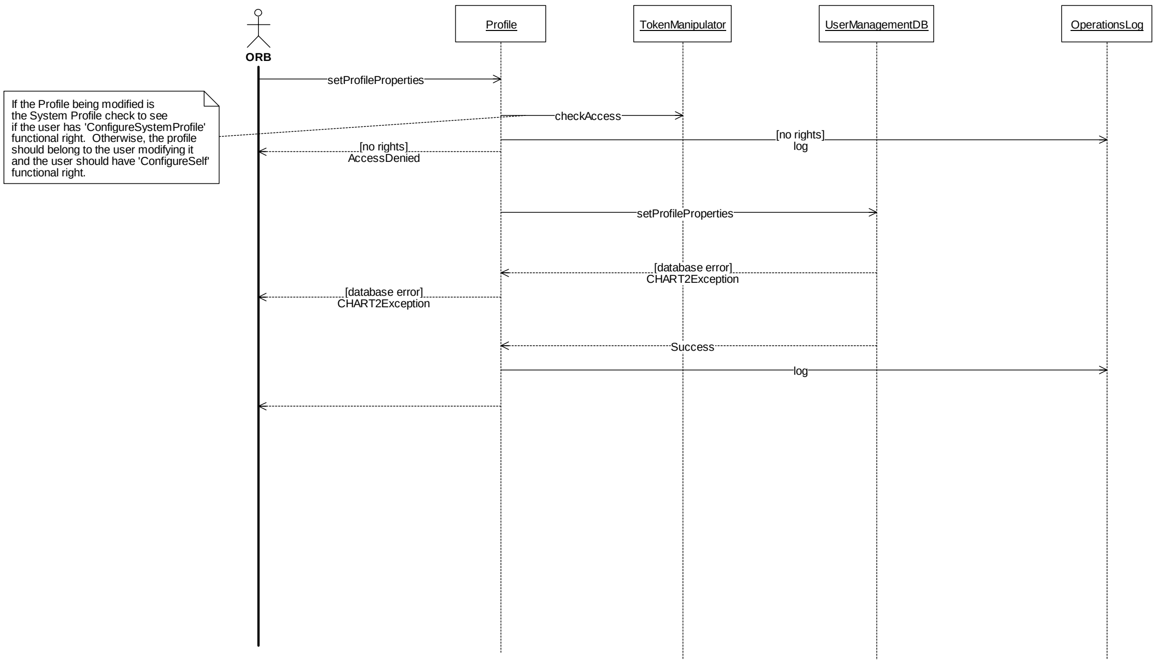

Figure 213. UserManagementModule:SetProfileProperties (Sequence Diagram)

3.19.2.14 UserManagementModule:SetRoleFunctionalRights (Sequence Diagram)

A user with proper functional rights may set the list of Functional Rights belonging to a role. Note that at the completion of this sequence the role will only have the rights that were set by this call.

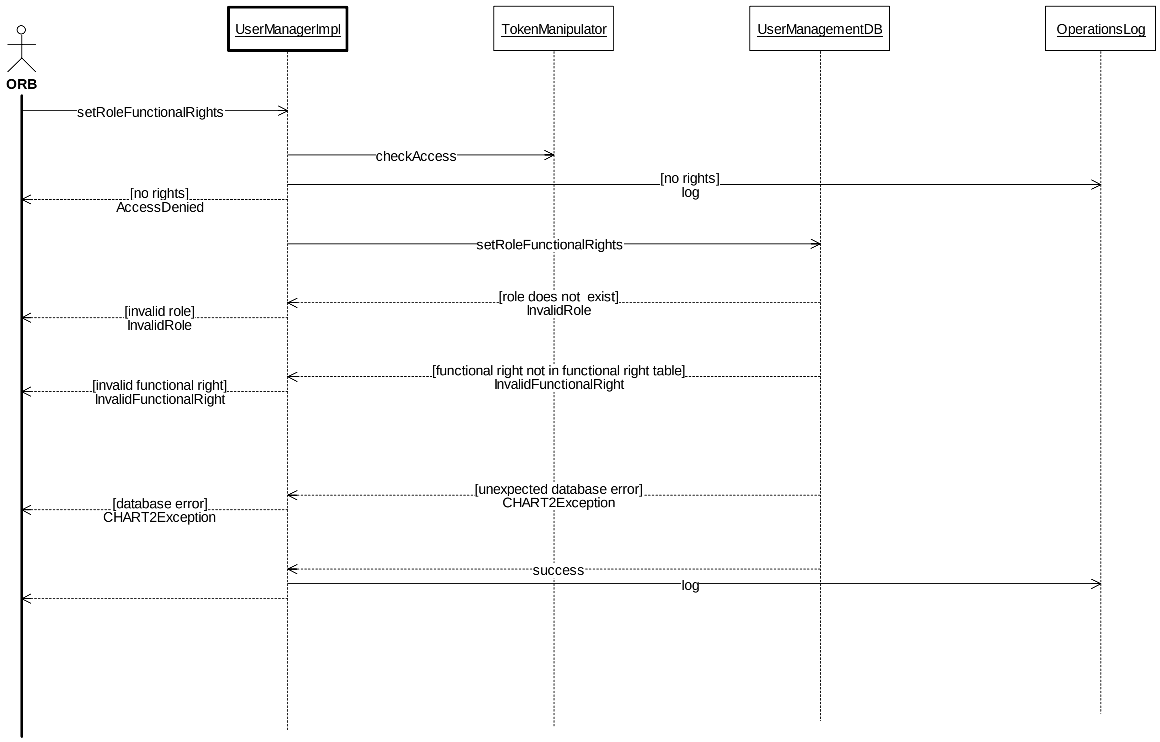

Figure 214. UserManagementModule:SetRoleFunctionalRights (Sequence Diagram)

3.19.2.15 UserManagementModule:SetUserPassword (Sequence Diagram)

A user with the proper functional rights may set the password that a user must specify in order to log into the system. This action does not require that the administrator be able to supply the users current password and, therefore, is restricted to administrative users. This function is included to deal with situations where users forget their system password.

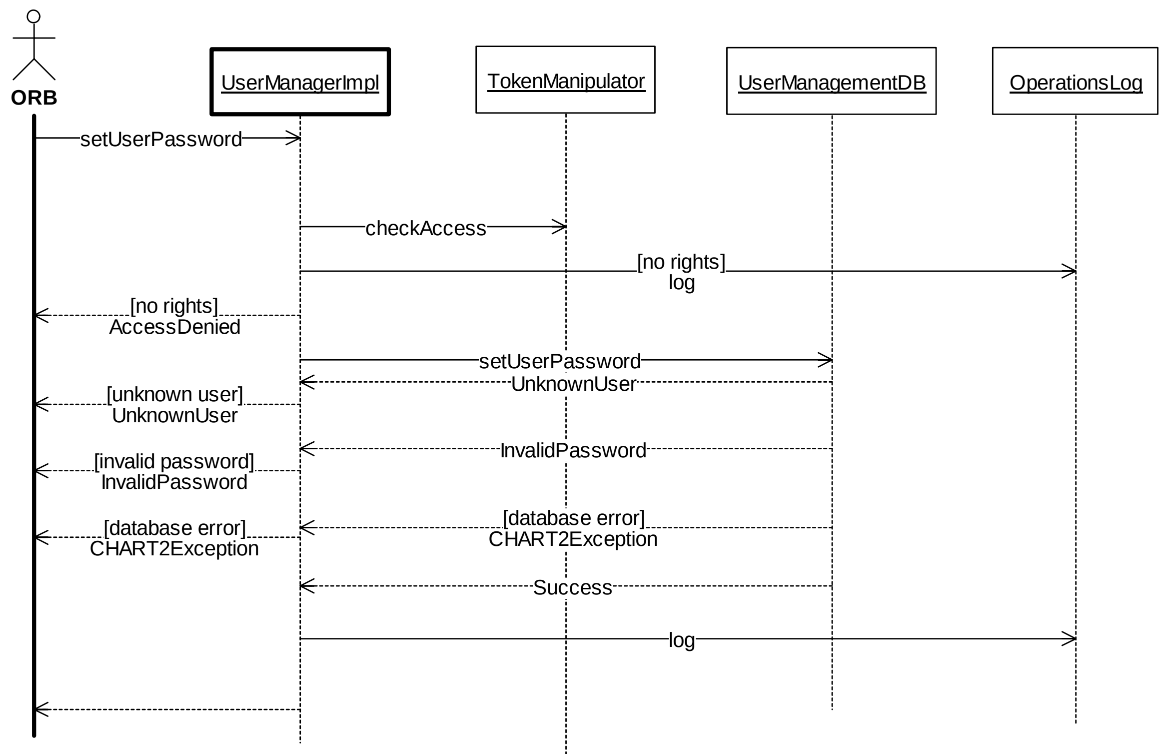

Figure 215. UserManagementModule:SetUserPassword (Sequence Diagram)

3.19.2.16 UserManagementModule:SetUserRoles (Sequence Diagram)

A user with the proper functional rights may assign set of roles to a user. The user will not get his/her new functional rights until he/she logs off and logs back on. Note that at the end of this operation the user will have only the roles assigned by this operation.

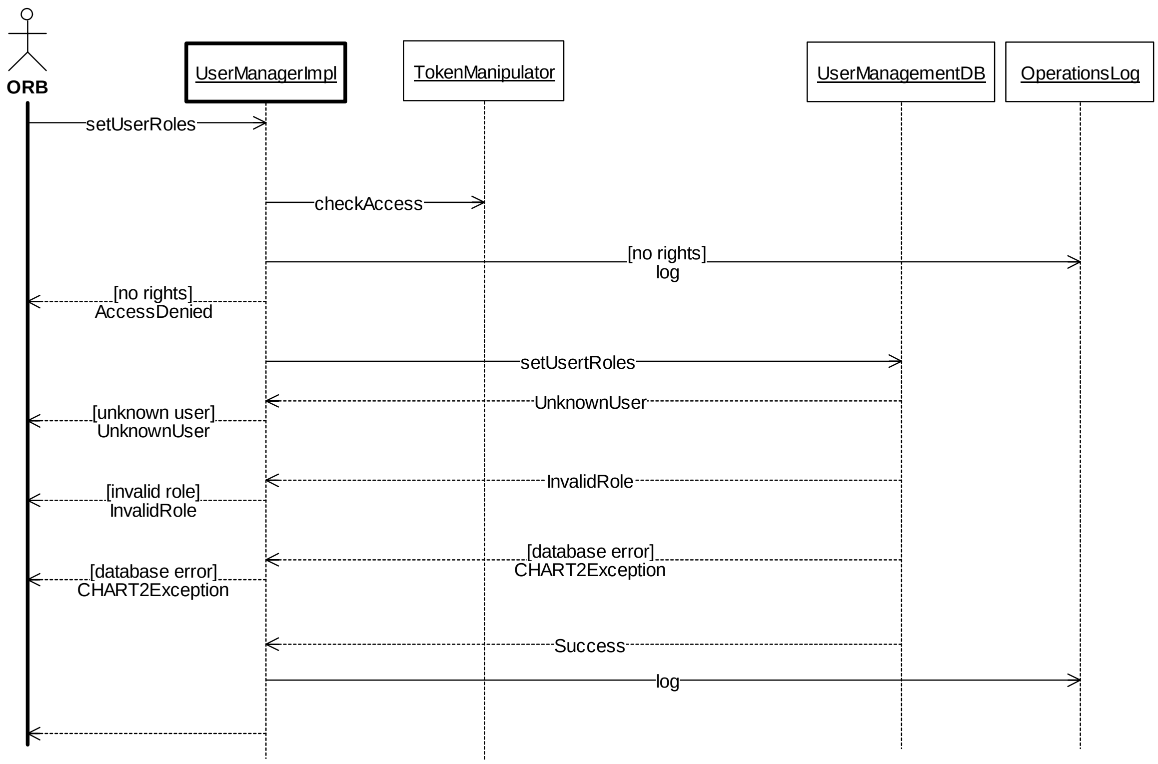

Figure 216. UserManagementModule:SetUserRoles (Sequence Diagram)

3.19.2.17 UserManagementModule:Shutdown (Sequence Diagram)

The user management module will withdraw the user management object from the trader, deactivates it from the POA and delete it.

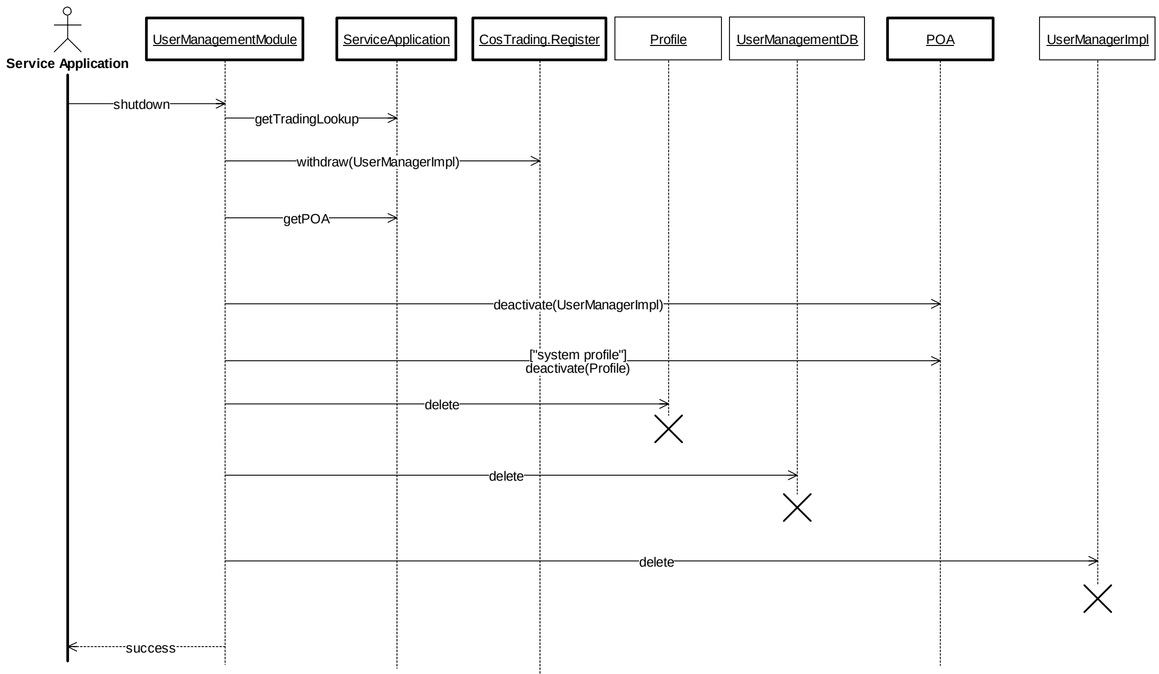

Figure 217. UserManagementModule:Shutdown (Sequence Diagram)

3.20 Utility

3.20.1 Classes

3.20.1.1 UtilityClasses (Class Diagram)

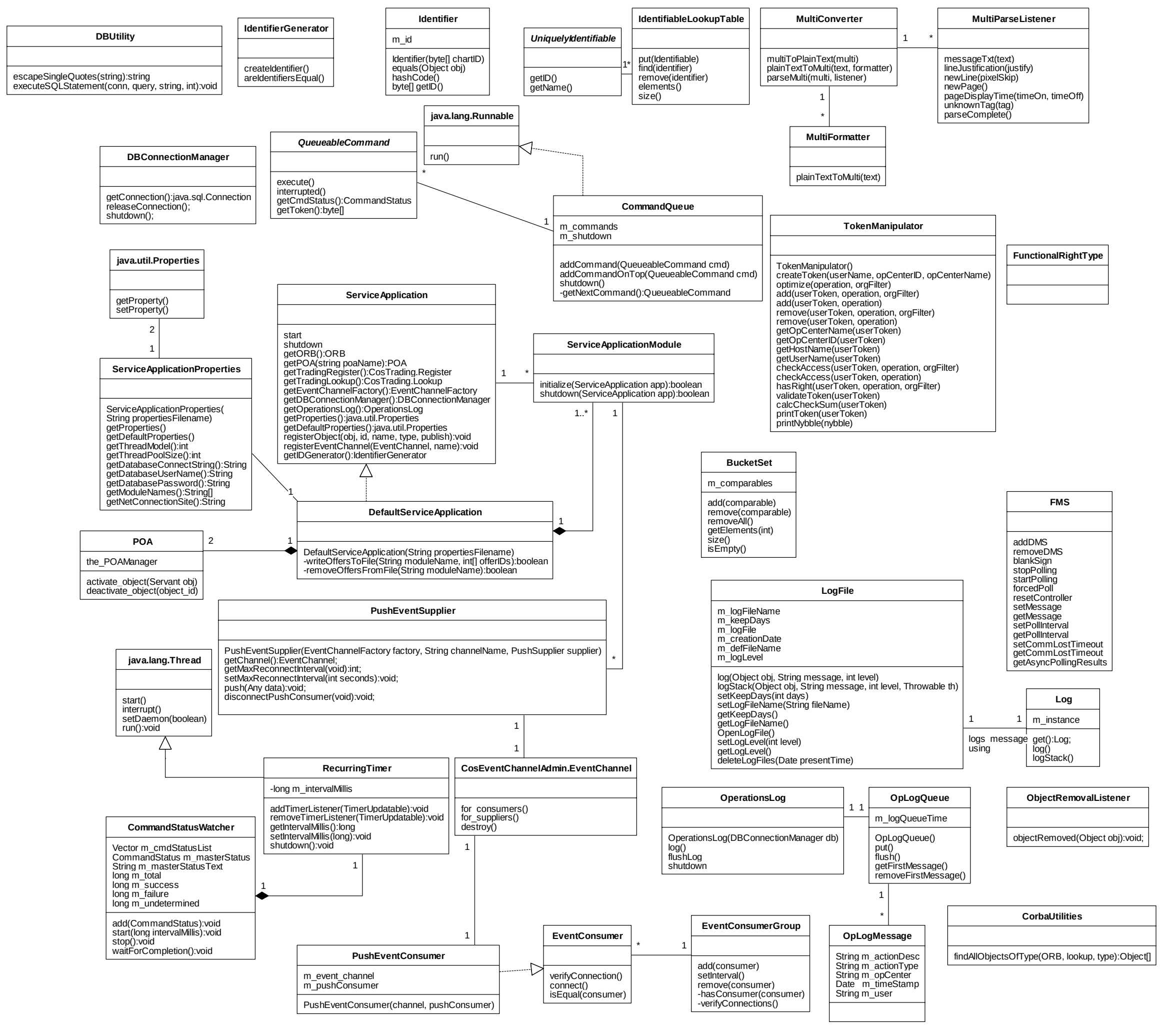

Figure 218. UtilityClasses (Class Diagram)

3.20.1.1.1 BucketSet (Class)

This class is designed to contain a collection of comparable objects. All of the objects added to this collection must be of the same concrete type. Each element in the collection has an associated counter that tracks how many times this element has been added. It is then possible to get only the elements which have been added to the collection n times where n is a positive integer value. This class is very useful for creating GUI menu's for multiple objects as it allows all objects to insert their menu items and then allows the user to get only those items that all objects inserted.

3.20.1.1.2 CommandQueue (Class)

The CommandQueue class provides a queue for QueuableCommand objects. The CommandQueue has a thread that it uses to process each QueuableCommand in a first in first out order. As each command object is pulled off the queue by the CommandQueue's thread, the command object's execute method is called, at which time the command performs its intended task.

3.20.1.1.3 CommandStatusWatcher (Class)

This class is a utility that monitors one or more command status objects for completion. It periodically checks each command status object's completion code and maintains statistics on the number of failures and successes. It provides a blocking method that waits for all command status objects to complete.

3.20.1.1.4 CosEventChannelAdmin.EventChannel (Class)

The event channel is a service that decouples the communication between suppliers and consumers of information.

3.20.1.1.5 CorbaUtilities (Class)

This class is a collection of static CORBA utility methods that can be used by both server and GUI for CORBA Trader service transactions.

3.20.1.1.6 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two separate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.20.1.1.7 DBUtility (Class)

This class contains methods that allow interaction with the database.

3.20.1.1.8 DefaultServiceApplication (Class)

This class is the default implementation of the ServiceApplication interface. This class is passed a properties file during construction. This properties file contains configuration data used by this class to set the ORB concurrency model, determine which ORB services need to be available, provide database connectivity, etc. The properties file also contains the class names of service modules that should be served by the service application. During startup, the DefaultServiceApplication instantiates the service application module classes listed in the properties file and initializes each.

The DefaultServiceApplication maintains a file of offers that have been exported to the Trading Service. Each module must provide an implementation of the getOfferIDs method and be able to return the offer ids for each object they have exported to the trader during their initialization. The DefaultServiceApplication stores all offer IDs in a file during its startup. Each module is expected to remove its offers from the trader during a shutdown. If the DefaultServiceApplication is not shutdown properly, it uses its offer ID file to clean-up old offers prior to initializing modules during its next start. This keeps multiple offers for the same object from being placed in the trader.

3.20.1.1.9 EventConsumer (Class)

This interface provides the methods that any EventConsumer object that would like to be managed in an EventConsumerGroup must implement.

3.20.1.1.10 EventConsumerGroup (Class)

This class represents a collection of event consumers that will be monitored to verify that they do not lose their connection to the CORBA event service. The class will periodically ask each consumer to verify its connection to the event channel on which it is dependent to receive events.

3.20.1.1.11 FMS (Class)

This class represents the CHART II system’s interface to the FMS SNMP manager. Most methods included in this class have an associated method in the FMS SNMP Manager DLL provided by the FMS Subsystem. The other methods in this class exist to provide easier interface to the DLL. As an example, this class contains a blankSign method that actually calls setMessage on the FMS Subsystem with the message set to blank and beacons off.

3.20.1.1.12 FunctionalRightType (Class)

This class acts as an enumuration that lists the types of functional rights possible in the CHART2 system. It contains a static member for each possible functional right.

3.20.1.1.13 IdentifiableLookupTable (Class)

This class uses a hash table implementation to store Identifiable objects for fast lookups.

3.20.1.1.14 Identifier (Class)

Wrapper class for a CHART2 identifier byte sequence. This class will be used to add identifiable objects to hash tables and perform subsequent lookup operations.

3.20.1.1.15 IdentifierGenerator (Class)

This class is used to create and manipulate identifiers that are to be used in Identifiable objects.

3.20.1.1.16 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

3.20.1.1.17 java.lang.Thread (Class)

This class represents a java thread of execution.

3.20.1.1.18 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its "defaults"; this second property list is searched if the property key is not found in the original property list.

3.20.1.1.19 Log (Class)

Singleton log object to enable applications to easily create and utilize a LogFile object for system trace messages.

3.20.1.1.20 LogFile (Class)

This class creates a flat file for writing system trace log messages and purges them at user specified interval. The log files created by this class are used for system debugging and maintenance only and are not to be confused with the system operations log that is modeled by the OperationsLog class.

3.20.1.1.21 MultiConverter (Class)

This class provides methods that perform conversions between the DMS MULTI mark-up language and plain text. It also provides a method that will parse a MULTI message and inform a MultiParseListener of elements found in the message.

3.20.1.1.22 MultiFormatter (Class)

This interface must be implemented by classes which convert plain text DMS messages to MULTI formatted messages.

3.20.1.1.23 MultiParseListener (Class)

A MultiParseListener works in conjunction with the MultiConverter to allow an implementing class to be notified as parsing of a MULTI message occurs. An exemplary use of a MultiParseListener would be the MessageView window that will need to have the MULTI message parsed in order to display it as a pixmap.

3.20.1.1.24 ObjectRemovalListener (Class)

This interface is implemented by objects that wish to be notified of objects being removed from the system. This is typically used by objects that store a collection of other objects, such as a factory, to allow them to remove objects from their collection when the object is to be removed from the system.

3.20.1.1.25 OperationsLog (Class)

This class provides the functionality to add a log entry to the CHART II operations log. At the time of instantiation of this class, it creates a queue for log entries. When a user of this class provides a message to be logged, it creates a time-stamped OpLogMessage object and adds this object to the OpLogQueue. Once queued, the messages are written to the database by the queue driver thread in the order they were queued.

3.20.1.1.26 OpLogQueue (Class)

This class is a queue for messages that are to be put into the system's Operations Log. Messages added to the queue can be removed in FIFO order.

3.20.1.1.27 OpLogMessage (Class)

This class holds data for a message to be stored in the system's Operations Log.

3.20.1.1.28 POA (Class)

This interface represents the portable object adapter used to activate and deactivate servant objects.

3.20.1.1.29 PushEventConsumer (Class)

This class is a utility class that will be responsible for connecting a consumer implementation to an event channel, and maintaining that connection. When the verifyConnection method is called, this object will determine if the channel has been lost and will attempt to re-connect to the channel if it has.

3.20.1.1.30 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier’s push rate.

3.20.1.1.31 QueueableCommand (Class)

A QueueableCommand is an interface used to represent a command that can be placed on a CommandQueue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed. This interface must be implemented by any device command in order that it may be queued on a CommandQueue. The CommandQueue driver calls the execute method to execute a command in the queue and a call to the interuppted method is made when a CommandQueue is shut down.

3.20.1.1.32 RecurringTimer (Class)

A recurring timer is a thread that notifies each TimerUpdatable object that has been registered on a specified period.

3.20.1.1.33 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHARTII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.20.1.1.34 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.20.1.1.35 ServiceApplicationProperties (Class)

This class provides methods that allow the DefaultServiceApplication to access the necessary properties from the java properties configuration file. It also provides a default properties file which can be retrieved by anyone holding a ServiceApplication interface reference. This gives each installed service module the opportunity to load default values before retrieving property values from the properties file.

3.20.1.1.36 TokenManipulator (Class)

This class contains all functionality required for user rights in the system. It is the only code in the system that knows how to create, modify and check a user's functional rights. It encapsulates the contents of an octet sequence that will be passed to every secure method. Secure methods should call the checkAccess method to validate the user. Client processes should use the check access method to verify access and optimize to reduce reduce the size of the sequence to only those rights which are necessary to invoke the secure method. The token contains the following information. Token version, Token ID, Token Time Stamp, Username, Op Center ID, Op Center IOR, functional rights

3.20.1.1.37 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

1.1.1.1 UtilityClasses2 (Class Diagram)

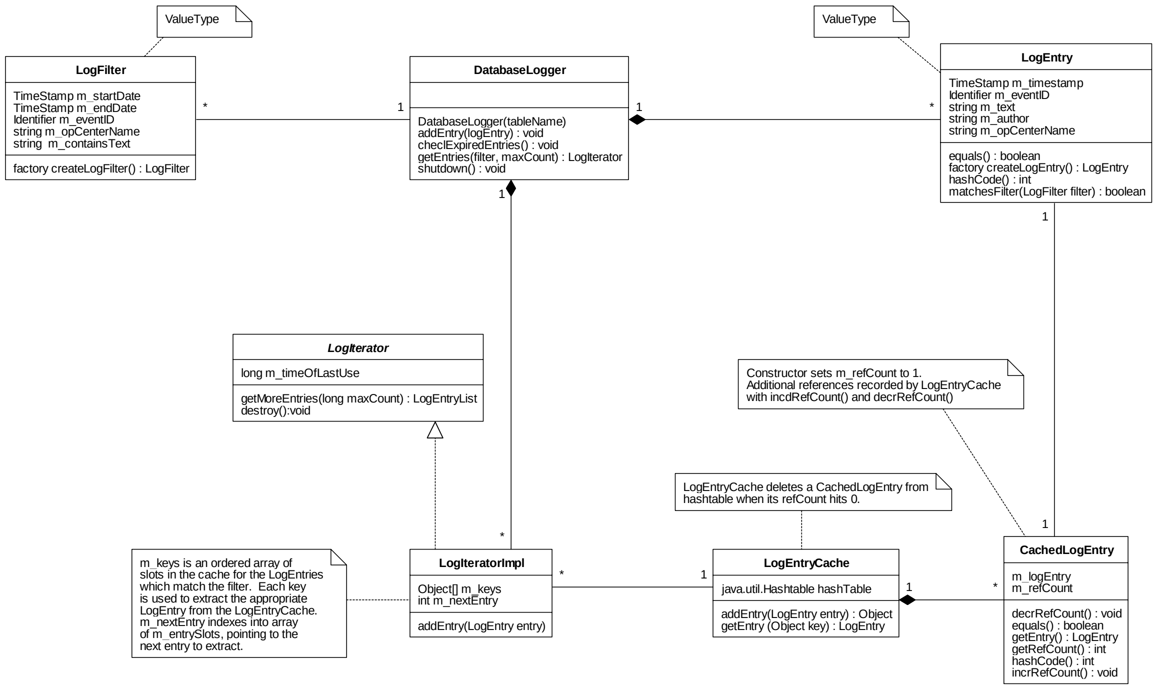

Figure 219. UtilityClasses2 (Class Diagram)

3.20.1.1.38 CachedLogEntry (Class)

This class represents a reference-counting object stored in a memory-efficient LogEntryCache. The object of this class encapsulates the stored log entry and adds a reference count.

3.20.1.1.39 DatabaseLogger (Class)

This class represents a generic database logger that can be used to log and retrieve information from the database. This class also provides a mechanism for the user to filter and retrieve logs that meet specific criteria.

3.20.1.1.40 LogEntry (Class)

This class represents a typical log entry that is stored in the database. This can be a general Communications Log entry or it can be a historical entry for a Traffic Event. Some Traffic Event actions (opening, closing, etc.) are logged in the Communications Log as well as in the history of the specific Traffic Event.

3.20.1.1.41 LogEntryCache (Class)

The LogEntryCache caches log entries returned from a database query which are in excess of the requestor-specified maximum number of entries to return at one time. The LogIterator stores references to the LogEntry objects thus cached, and requests additional objects as needed. The LogEntryCache uses reference counting to prevent storing duplicate copies of LogEntry objects, and it deletes LogEntry objects when they are no longer needed.

3.20.1.1.42 LogFilter (Class)

This class is used to specify the criteria to be used when getting entries from the Communications Log. The caller would create an object of this type specifying the criteria that each log entry must match in order to be returned.

3.20.1.1.43 LogIterator (Class)

This class represents an iterator to iterate through a collection of log entries. If a retrieval request results in more data than is reasonable to transmit all at once, one clump of entries is returned at first, together with a LogIterator from which additional data can be requested, repeatedly, until all entries are returned or the user cancels the operation.

3.20.1.1.44 LogIteratorImpl (Class)

The LogIteratorImpl implements the LogIterator interface; that is, it does the actual work which clients can request via the LogIterator interface. The LogIteratorImpl stores data relating to cached LogEvents for a single retrieval request, and implements the client request to get additional clumps of data pertaining to that request.

3.20.2 Sequence Diagrams

3.20.2.1 DatabaseLogger:getEntries (Sequence Diagram)

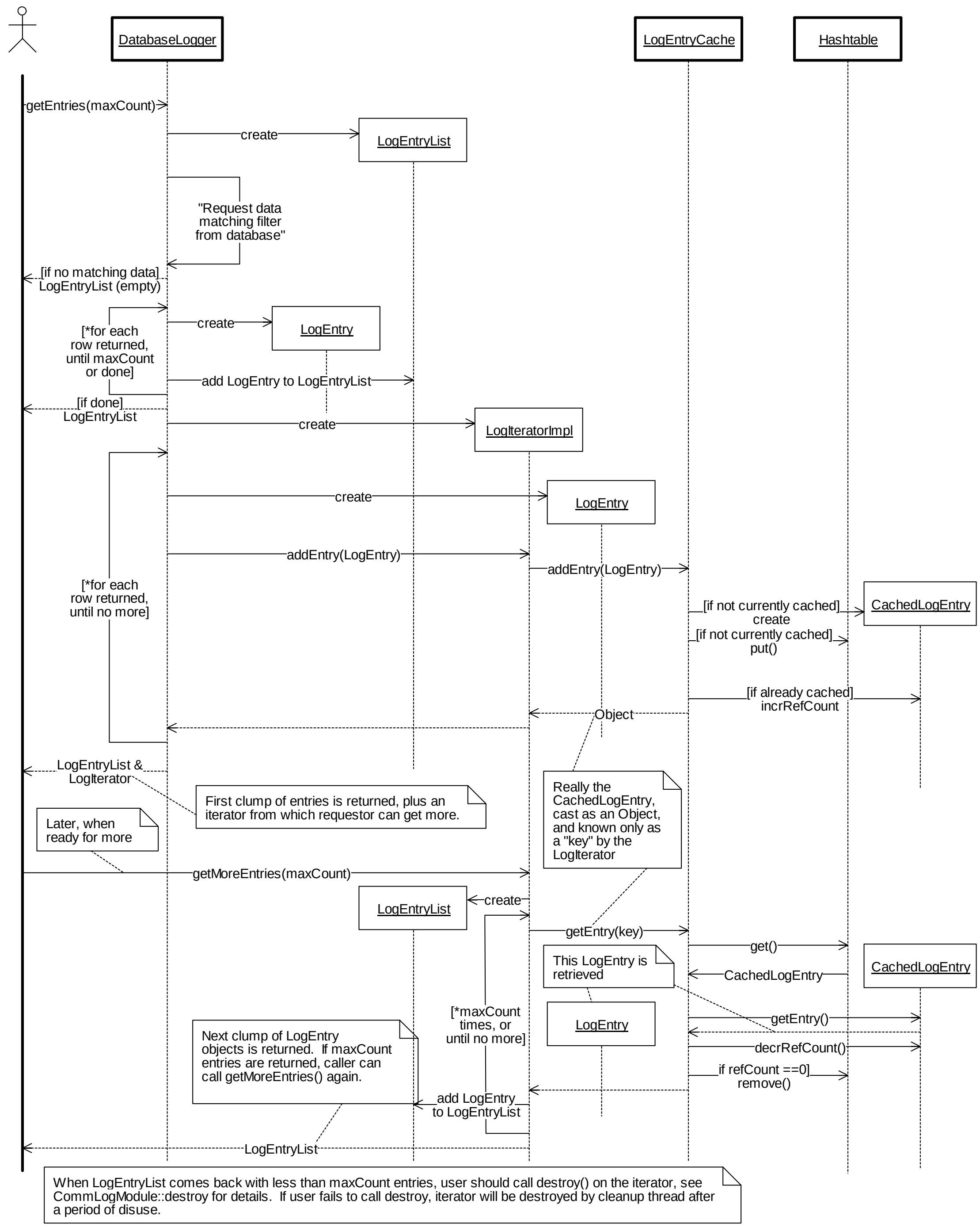

Figure 220. DatabaseLogger:getEntries (Sequence Diagram)

3.20.2.2 DictionaryWrapper:checkForBannedWords (Sequence Diagram)

This diagram shows processing performed by the DictionaryWrapper that is representative of all methods that it duplicates in the Dictionary interface. When a method is called that is to be delegated to a system dictionary, the DictionaryWrapper first attempts to use the dictionary references (if any) that it has already discovered during a previous method invocation. If no references exist (this is true for the first usage of the wrapper) or if all existing references return CORBA failures when used, the DictionaryWrapper queries the trader for all Dictionaries in the system and then attempts to use each until a “live” reference is found or all of the newly discovered references return CORBA failures when used.

A timestamp is used to prevent a flurry of trader queries when no Dictionary objects are available. Prior to doing a trader query to (re)discover dictionaries, the DictionaryWrapper makes sure that at least a minimum amount of time has elapsed since the last time it tried to find a dictionary. The use of synchronization around the discovery process also helps to prevent a flood of trader queries.

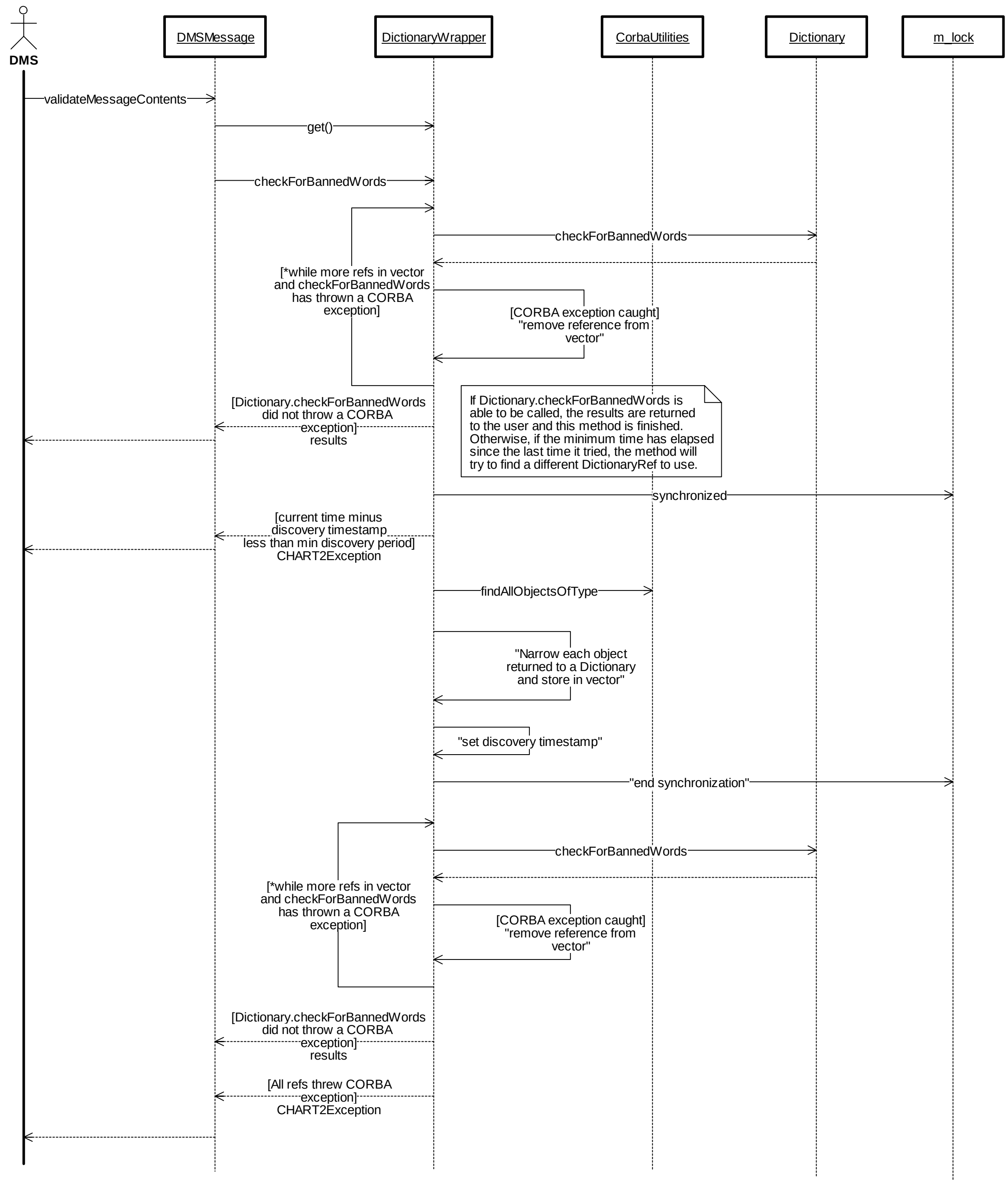

Figure 221. DictionaryWrapper:checkForBannedWords (Sequence Diagram)

Acronyms

The following acronyms appear throughout this document:

API	Application Program Interface
BAA	Business Area Architecture
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DMS	Dynamic Message Sign
DTMF	Dual Tone Multiple Frequency
EORS	Emergency Operations Reporting System
FMS	Field Management Station
GUI	Graphical User Interface
HAR	Highway Advisory Radio
IDL	Interface Definition Language
ITS	Intelligent Transportation Systems
LATA	Local Access and Transport Areas
NTCIP	National Transportation Communications for ITS Protocol
OMG	Object Management Group
ORB	Object Request Broker
POA	Portable Object Adapter
R1B2	Release 1, Build 2 of the CHART II System
TTS	Text To Speech
UML	Unified Modeling Language

References

CHART II Release 4 Interim BAA Report, document number M361-BA-004R0, Computer Sciences Corporation and PB Farradyne.

CHART II System Requirements Specification Release 1 Build 2, document number M361-RS-002R1, Computer Sciences Corporation and PB Farradyne.

R1B2 High Level Design, document number M362-DS-005R0, Computer Sciences Corporation and PB Farradyne.

FMS R1B1 High Level Design, document number M303-DS-001R0, Computer Sciences Corporation and PB Farradyne.

CHART II to Field Management Station (FMS) Interface Control Document (ICD), document number M361-ID-001R0, Computer Sciences Corporation and PB Farradyne.

The Common Object Request Broker: Architecture and Specification, Revision 2.3.1, OMG Document 99-10-07

Martin Fowler and Kendall Scott, *UML Distilled,* Addison-Wesley, 1997

TELE-SPOT 3001 Sign Controller Communications Protocol, document number 750208-040 v2.3, T-S Display Systems Inc., 1995

Functional Specification for FP9500ND – MDDOT Display Control System, document number A316111-080 Rev. A6, MARK IV Industries Ltd., 1998.

Maintenance Manual for the FP1001 Display Controller, document number 316000-443 Rev. E, Ferranti-Packard Displays, 1987

FP2001 Display Controller Application Guide, document number A317875-012 Rev. 8, F-P Electronics, 1991

Engineering Specification - Brick Sign Communications Protocol, Rev. 1, ADDCO Inc., 1999.

PCMS Protocol version 4, document number 32000-150 Rev. 5, Display Solutions, 2000

BSC Protocol Specification (Data Link Protocol Layer), v. 1.3, Fiberoptic Display Systems Inc., 1996

Sylvia Variable Message Sign, Command Set 9403-1, v. 1.4, Fiberoptic Display Systems Inc., 1996

2.5 Mile AM Travelers Information Station Instruction Manual For: Maryland State Highway Administration, Information Station Specialists.

Technical Practice RC-2A Remote Touch-Tone On/Off Industrial Controller, Viking Electronics Inc., August 1993.

Appendix A – Functional Rights

This table lists the functional rights that exist in the CHART II system and the operations to which they grant access.

Functional Right Required	Operation	Organization Filterable
BasicOperations	Add Comm Log entries	No
	Get Comm Log entries	No
ConfigureDMS	Add DMS	Yes
	Remove DMS	Yes
	Set DMS Configuration	Yes
ConfigureDMS or ViewDMSConfig	Get DMS Configuration	Yes
ConfigureHAR	Add HAR	Yes
	Add SHAZAM	Yes
	Remove HAR	Yes
	Remove SHAZAM	Yes
	Set HAR Associated with a Notifier(DMS or SHAZAM)	Yes
	Set HAR Configuration	Yes
	Set HAR Message Notifier(DMS or SHAZAM) Direction	Yes
	Set SHAZAM Configuration	Yes
ConfigureHAR or ViewHARConfig	Get HAR Configuration	Yes
	Get SHAZAM Configuration	Yes
ConfigureSelf	Get User Profile	No
	Set User Password	No
ConfigureSelf, ConfigureSystemProfile	Delete Profile Properties	No
	Set Profile Properties	No
ConfigureTrafficEvent	Add Traffic Event Log Entry	No
	Associate Event	No
	Change Event type	No
	Check if Congestion Event is a Recurring event	No
	Close Traffic Event	No
	Override Incident Lane Open Close Time	No
	Override Traffic Event Closure Time	No
	Set Congestion Event as a Recurring event	No
	Set Incident Road Conditions	No
	Set Incident Type	No

Functional Right Required	Operation	Organization Filterable
	Set Incident Vehicle Data	No
	Set Roadway Event lane configuration	No
	Set Traffic Event as Primary event	No
	Set Traffic Event as Secondary event	No
ConfigureUsers	Change User Password	No
	Create Role	No
	Create User	No
	Delete Role	No
	Delete User	No
	Grant Role	No
	Revoke Role	No
	Set Role Functional Rights	No
	Set User Roles	No
Maintain DMS	Blank DMS	Yes
	Perform DMS Pixel Test	Yes
	Perform DMS Test	Yes
	Poll DMS	Yes
	Reset DMS Controller	Yes
	Set DMS Message	Yes
MaintainHAR	Blank HAR	Yes
	Delete HAR Slot Message	Yes
	Refresh SHAZAM	Yes
	Reset HAR	Yes
	Set HAR Message	Yes
	Set HAR Transmitter Off	Yes
	Set HAR Transmitter On	Yes
	Set SHAZAM Beacons Off	Yes
	Set SHAZAM Beacons On	Yes
	Setup HAR	Yes
	Store HAR Slot Message	Yes
Manage Services	Shutdown Service	No
ManageDeviceComms	Put a device in Maintenance Mode	Yes
	Put a device Online	Yes
	Take a device Offline	Yes
ManageDictionary	Add a list of Approved Words to Dictionary	No
	Add a list of Banned Words from Dictionary	No

Functional Right Required	Operation	Organization Filterable
	Remove a list of Approved Words from Dictionary	No
	Remove a list of Banned Words from Dictionary	No
ManageDictionary or ViewDictionary	Get Approved Words from Dictionary	No
	Get Banned Words from Dictionary	No
ManageUserLogins	Force Logout	No
	Force Logout	No
ModifyMessageLibrary	Create Message Library	No
	Create Stored Message	No
	Remove Library	No
	Remove Stored Message	No
	Remove Stored Message	No
	Set Message associated with Stored Message	No
	Set Message Library Name	No
	Set Stored Message Data	No
ModifyPlans	Add Plan Item	No
	Create Plan	No
	Remove Plan	No
	Remove Plan Item	No
	Remove Plan Item	No
	Set Plan Item Data	No
	Set Plan Item Name	No
	Set Plan Name	No
Must pass the token of the user logging out	Change User	No
	Logout User	No
RespondToTrafficEvent	Add a message to Arbitration Queue	No
	Add Resource Response Participation	No
	Add Response Plan Item	No
	Execute Response Plan Item	No
	Execute Traffic Event Response	No
	Override Organization responded time	No
	Override Resource arrival time	No
	Override Resource departure time	No

Functional Right Required	Operation	Organization Filterable
	Remove a message from Arbitration Queue	No
	Remove Response Device	No
	Remove Response Participation	No
	Remove Response Plan Item	No
	Set Organization notification.	No
	Set Organization participation response to Event	No
	Set Resource arrived on scene	No
	Set Resource departed from scene	No
	Set Response Plan Item data	No
	Set Response Plan Item description	No
RespondToTrafficEvent, ViewTrafficEventData	Get Response Plan Item data	No
SetHARMessage	Activate HAR Message Notice	Yes
	Deactivate HAR Message Notice	Yes
	Set HAR message and Notifiers	Yes
TransferAnySharedResource	Clear Controlling Operations Center	Yes
	Set Controlling Operations Center	Yes
	Transfer Shared Resources	Yes
ViewUserConfig or ConfigureUsers	Get Role Functional Rights	No
	Get Roles	No
	Get User Roles	No
	Get Users	No
ViewUserLogins	Get Login Sessions	No

Appendix B – Glossary

Action Event	A Traffic Event related to the disposition of actions in response to device failures and non-blockage events (e.g. signals, debris, utility, and signs).
Approved Word	A word that is known to the system and has been approved for use when communicating with the motoring public via a messaging device. The dictionary will suggest words to the operator when it encounters a word that has not been previously approved.
Arbitration Queue	A prioritized queue containing messages for display or broadcast on a traveler information device.
Banned Word	A word that may not be used when communicating with the motoring public via a messaging device such as a HAR or DMS.
Comm Log	A collection of information received from any source that requires no action.
Congestion Event	A Traffic Event related to roadway congestion situations. Congestion Events may be recurring or non-recurring.
CORBA Event	A CORBA mechanism using which different CHART2 components exchange information without explicitly knowing about each other.
CORBA Trader	A CORBA service that facilitates object location and discovery. A server advertises an object in the Trading Service based on the kind of service provided by the object. A client locates objects of interest by asking the Trading Service to find all objects that provide a particular service.
Data Model	An object repository that keeps track of changes to the various objects in the repository and informs about these changes as they occur, to observers who are interested in the objects in the repository. A Data Model identifies the subject in a Subject/Observer design pattern.
Dictionary	A collection of banned and approved words.

Deployable Resource	Any resource that can be deployed to the scene in order to provide assistance during a traffic event.
DMS	A Dynamic Message Sign that can be controlled by one Operations Center at a time.
DMS Stored Message Item	A plan item that is used to set a specific message on a specific DMS when added to a Traffic Event response plan and activated.
Emergency Operations Reporting System	A system external to CHART II that (among other things) keeps track of planned roadway closures and permits.
Factory	A CORBA object that is capable of creating other CORBA objects of a particular type. The newly created object will be served from the same process as the factory object that creates it.
FMS	Field Management Station through which the CHART II system communicates with the devices in the field.
Functional Right	A privilege that gives a user the right to perform a particular system action or related group of actions. A functional right may be limited to pertain only to those shared resources owned by a particular organization or can pertain to the shared resources of all organizations.
Graphical User Interface	Part of a software application that provides a graphical interface to its user.
GUI Wrapper Object	A GUI wrapper object is one that wraps a server object to provide it with GUI functionality such as menu handling. It also helps in performance enhancement by caching data locally thereby avoiding network calls when not necessary.
HAR	A Highway Advisory Radio which can be controlled by one Operations Center at a time.
HAR Message	A message which is capable of being stored on a HAR. It is composed of a message header, body and footer.
HAR Message Clip	A message clip is part of a HAR message that could be a header

	or body or footer. It can be stored either as a text or in one of the binary forms (WAV, MP3 etc).
HAR Message Slot	A message slot is one of the numbered message stores inside the HAR device that can be used to store pre-fabricated messages useful for quick retrieval and playing.
Incident Event	A Traffic Event that is entered by an Operator in response to one of the following types of incidents: Disabled in roadway, Personal injury, Property damage, Fatality, Debris in roadway, Vehicle fire, Maintenance, Signal call, Police activities, Off-road activity, Declaration of emergency, Weather, or Other.
Installable Module	A plugable GUI module that provides a specific function, which when registered with the GUI is called on to initialize itself at the time of GUI startup and shut down at the time of GUI shut down.
Lane Closure	The closure of one or more roadway lanes resulting from a Traffic Event.
Message Library	A collection of stored messages that can be displayed on the DMS or broadcast on a HAR.
Navigator	A Navigator is a GUI window that contains a tree on the left-hand side and a list on the right hand side. Tree elements represent groups of objects and the list on the right hand side represents the objects in the selected group.
Object Discovery	A GUI mechanism in which the client periodically asks the CORBA Trading Service to find objects of those types that are of interest to the GUI, such as DMS, HAR, Plan etc.
Operations Center	A center where one or more users may log in to operate the CHART II system. Operations centers are assigned responsibility for shared resources that are controlled by users who are logged in at that operations center.
Operator	A CHART II user that works at an Operations Center.
Organization	An organization is an agency that participates in the CHART II system and owns one or more Shared Resources.

Plan	A collection of plan items that can be added to the response plan of a traffic event as a group.
Plan Item	An action in the system that can be set up in advance to be activated one or more times in the future. Plan items must be contained in a plan. Specific types of plan items exist for specific functionality. A plan item may be copied to a traffic event response plan and subsequently activated.
Response Plan	A collection of response plan items created in response to a traffic event that can be activated as a group..
Response Plan Item	An action in the system that can be set up in response to a traffic event. Response plan items must be contained in a response plan. Specific types of response plan items exist for specific functionality. A response plan item carries out its specific task when activated
Role	A Role is a collection of functional rights that a user may perform. The roles that pertain to a particular user for a particular login session are determined when he/she logs into the system.
Safety Message Event	A Traffic Event that is entered by an Operator to display and/or broadcast safety messages.
Service Application	A software application that can be configured to run one or more service application modules and provides them basic services needed to serve CORBA objects.
Service Application Module	A software module that serves a related group of CORBA objects and can be run within the context of a service application.
Shared Resource	A resource that is owned by an organization. A user may be granted access to a shared resource owned by an organization through the functional rights scheme.
SHAZAM	A device used to notify the traveling public of the broadcast of a HAR message.

Sign	see DMS
Stored Message	A message that may be broadcast on a HAR or displayed on a DMS.
System Profile	Information used to define the configuration of the system. Properties stored in the system profile apply to all users when they are logged in.
Token	A token or access token is a security blob that encloses information about a user and the functional rights associated with the user. All secured CHART2 operations require a token to be passed to it and based on the functional rights found in a token a user is allowed or denied access.
Traffic Event	A traffic event represents a roadway event that is affecting traffic conditions and requires action from system operators.
Transferable Shared Resource	A shared resource that can be transferred from one operations center to another by a user with the appropriate functional rights.
User	A user is somebody who uses the CHART II system. A user can perform different operations in the system depending upon the roles they have been granted.
User Profile	A set of information used to correctly configure an individual user's GUI on startup.
Weather Service Alert Event	A Traffic Event that is entered by an Operator in response to National Weather Service advisories.